Carsten Hood
CSCE 221H-200
April 8, 2014

Program 3 Report
Sorting Algorithms

Introduction

This report provides an analysis of the theoretical and experimental performances of various sorting algorithms. It presents the setup and results of tests designed to demonstrate the algorithms' complexities and provides a discussion of their results. Three priority queue sorts, insertion sort, selection sort, and heap sort, are studied, as well as merge sort, radix sort, and three implementations of quick sort with differing methods of pivot selection. All algorithms are tested using varying input sequence orders and a spread range of input sizes.

Theoretical Analysis

The priority queue selection sort algorithm runs independently of the order of the input sequence. Because of this, the best, worst, and average case complexities are identical. To choose the minimum element, all remaining elements must be examined. Since there will be one less element with each successive iteration, and there will be n iterations total, we have: $(n - 1) + (n - 2) + ... + 1 = n(n - 1)/2 = O(n^2)$.

Insertion sort is similar, but the running time is influenced by the order of the input structure. The algorithm only scans through as many elements as necessary to place the next one in the output sequence. Therefore, if the input sequence is already sorted, insertion sort runs in O(n). This is the best case; only one comparison is made for each element. The average case is $O(n^2)$; the algorithm will likely need to compare each of n elements with some fraction of elements in the output sequence. $n*O(n) = O(n^2)$. In the worst case the sequence is in reverse order. Each element must be compared with every previously sorted element before it is placed in the sequence. $1 + 2 \ldots + (n - 2) + (n - 1) = n(n - 1) / 2 = O(n^2)$.

Heap sort is a priority queue sort using a heap structure. First, the data is built into a heap. With each insertion, the heap order must be maintained. This takes amortized linear time: as more elements are sorted, less up-heap calls are necessary. Then each successive minimum element is removed. Maintaining heap order involves a down-heap operation. This is bounded by the height of the heap, as an element will need to be moved through each level to the bottom of the heap in the worst case. The height h is O(logn). Overall: $O(n) + n*O(logn) = O(nlogn)$. This is the best, average, and worst case for heap sort; no special input sequence will alter the complexity.

Merge sort uses the divide-and-conquer paradigm. It splits an input sequence in half and recursively calls the algorithm on these two subsequences. Then it merges the

subsequences. The recurrence equation of the form T(n) = D(n) + k T(n/c) + C(n) becomes T(n) = D(n) + 2T(n/2) + M(n). Dividing the sequence D(n) takes O(1) and combining M(n) takes O(n) using a merge algorithm. We have: 2T(n/2) + O(n) = O(n) + 2O(n/2) + … + n*O(1) = O(nlogn). Merge sort runs in time independent of the input sequence, so worst, best, and average cases have equal efficiencies. Every element must be examined during the merge step, and there must be O(logn) steps total.

The complexity of quick sort depends on its pivot selection. The worst case is that an extreme pivot is chosen in each step. Then only one value, the pivot, is eliminated from further sorting. In this case n steps are taken, and each remaining value is compared to the pivot in each step. The recurrence relation is T(n) = T(n-1) + T(1) + O(n). This simplifies to O(n^2). If the last element is always chosen as the pivot, the worst case occurs when the sequence is already sorted. With random pivot selection, this could result from bad luck. However, this case is avoided by choosing the median of three or more elements distributed across the sequence. In the best case of quick sort the sequence is always split in half. When the last element is used for pivot selection, this occurs when the last element of each subsequence is the median element for that subsequence. This case would be extremely unlikely in practice. It has the recurrence relation T(n) = 2T(n/2) + O(n) which simplifies to O(nlogn). However, even when this ideal scenario is not reached, as long as pivots are chosen that usually split the sequence into fractions, quick sort runs in O(nlogn). This is the best and average case, and is likely when choosing a random pivot, guaranteed by choosing a median pivot, and possible using the last element as the pivot when the sequence is not sorted.

Radix sort's complexity is always the same. For every key, or dimension d, the algorithm sorts all n elements into buckets, loops through all N buckets, and then copies the n elements back into the sequence.. Therefore the running time is d*(O(n)+O(N)+O(n)) = O(d(n+N)).

Experimental Setup

The experiments described in this report were performed on a Mac computer running OS X 10.9.1 with a 2.7 GHz Intel Core i7 processor and 8GB RAM memory. C++ code was compiled using Apple LLVM 5.0.

The standard library ctime was used to time the sorting operations. Executions were repeated at least four times as necessary using the formula: *iterations* = 4 * (*max_size*)/*size* where *max_size* = 2^14 = 16384. Using this formula, 32768 executions were averaged for the smallest input size of 2 to compensate for clock inaccuracy. Input sizes greater than 16384 triggered program failure for several sorting algorithms on the computer used for testing. Therefore, 16384 was chosen as the maximum input size and 2 as the minimum in order to cover a wide range of inputs.

After each sort operation the sequence was assigned to a copy of the original sequence in order to be reset for the next sorting. This assignment operation takes linear time, as each element must be deleted and then copied. While this may have influenced the experimental

results, the O(n) cost of the assignment can be discounted in calculation and analysis over a range of sizes because every sort takes greater asymptotic time than O(n). The O(n) time of the copy is cancelled by the O(d(n+N)), O(nlogn), or O(n^2) time of the sorting operations.

Integers were chosen as the sequence elements for the simplicity of their operations, such as the comparisons involved in sorting, and the ease of their random generation. Integers use little space in memory, which minimizes the time required to reset a sequence after each sort. Also, the implementation used for radix sort requires integer values.

A function generated input sequences given a size and a desired ordering dictated by a string parameter. Sequences could be in sorted order, reverse order, contain a series of identical elements, or otherwise be randomly populated with integer values between 0 and the sequence size. A special sequence was generated to provide the highly unlikely best-case scenario for the last-element-pivot implementation of quick sort; the medians of the sequence and each subsequence were moved to the ends of their corresponding sequences, optimizing pivot selection for the sort algorithm. This sequence also provided an interesting test case for the other quick sort algorithms.

Each sort algorithm was assigned three input sequence orderings to represent its best, worst, and average cases. A random ordering was always used for the average case. This input provides the most balanced comparison between all the algorithms. Insertion sort was tested with a sorted sequence for the best case, and a reversed sequence for its worst case. Quick sort was tested with an initially sorted sequence as its worst case. For the sorting algorithms with no asymptotic differences for any input, sequences were chosen to provide comparison with other algorithms or to test interesting scenarios. Radix sort was tested once with a sequence of zeroes, a special case because all elements are placed in the same bucket. Merge sort and heap sort, which share consistent complexities, were tested with pre-sorted, same-element, and randomized input sequences.

Experimental Results

Figures 2A through 2F plot the running times of individual sorting algorithms with different input sequences. Figures 3A through 3F provide corresponding plots of the average case input for each algorithm. Time measurements are divided by the expected time to provide more insight into the relation between the experimental results and theoretical analysis. In these plots the initial values for small sizes are higher but eventually level out into nearly linear curves. These horizontal plots suggest that results match analysis and this can be used to determine the Big-O constants for each experiment.

Insertion sort (Figures 2A & 3A) is demonstrated to have a best-case complexity of O(n), while the random sequence, representing the average case, clearly depicts the O(n^2) worst and average case complexities. Interestingly, the input sequences in ascending and descending order both appear to cause linear running times. This contradicts my expectation that a descending sequence represents the worst case for insertion sort. After studying the implementation, I determined that the descending order should actually be the best input, and the ascending sequence the worst. However, this still does not explain

the results. I checked that my implementation and test cases were correct and could not find any answer as to why insertion sort performed linearly with both ordered and reverse-ordered sequences. Indeed, insertion sort proved to be the only algorithm tested with unexpected results. In Figure 3A when the time of the average case is divided by the expected time, O(n^2), a linear path is revealed after low input sizes. Big-O constants of the form (k0, C) are determined to be (32, 2.13E-08).

Figure 2B shows selection sort operating on the same three test cases as insertion sort. As expected, all three selection sort plots are nearly identical; selection sort always has a running time of O(n^2), which differs from insertion sort's best case running time of O(n). Figure 3B divides the time by the expected time, O(n^2), resulting in a linear path, which confirms the expected complexity. After elevated initial values, the path evens out after a size of n = 32. Therefore, the Big-O constants are determined to be n0 = 32 and C = 5.19E-08.

Three plots of heap sort are shown on Figure 2C. There are no best, worst, or average cases, as heap sort always runs in O(nlogn) time. The graphed lines are very similar, which is consistent with analysis. Figure 3C confirms heap sort's complexity, as the plot is nearly linear when the size n is divided by the expected time, nlogn. Suitable Big-O constants of the form (k0, C) are found to be (64, 1.12E-07).

Figure 2D plots merge sort over a range of input sequences. All plots are nearly identical. This matches the analysis that merge sort consistently runs in O(nlogn). Figure 3D's plot, taking expected time into account, shows a nearly linear line. Big-O constants are observed to be k0 = 8, C = 5.32E-07.

All three implementations of quick sort are represented in Figure 2E: last-element pivot selection, median-of-three pivot selection, and random pivot selection. For the implementation in which the last element of a subsequence is chosen as the pivot, the worst case runs in O(n^2) and occurs when the input sequence is already sorted in ascending or descending order. This case clearly has a quadratic curve on the plot and diverges from the other curves, confirming analysis. The intended best-case scenario, in which the input sequence was generated to have median elements at the end of each subsequence, appears to have failed in its purpose of providing the perfect running time; the random sequence is clearly more efficient. Despite my efforts, my sequence-generating algorithm likely did not work with the implementation. However, the intended ideal sequence and the random sequence both share the complexity O(nlogn). Theoretically this is the best and expected case. The other two implementations of quick short are plotted using random input sequences for comparison. All three quick sorts appear identical when sorting randomized input. However, review of the plotted values revealed that the quick sort implementation in which the pivot is randomly selected is more efficient for all but the largest two input sizes. Because of this, this implementation of quick sort is chosen to represent the quick sort algorithm in its comparison with the other sorts in Figure 1. In Figure 3E the Big-O constants for quick sort with last element pivot selection are C = 7.25504E-08 and n0 = 8. After sizes of 8, the plotted line runs near to but underneath the horizontal line at 7.25504E-08.

Figure 2F plots radix sort with three input sequences: a sorted sequence, a series of zeroes, and a randomized sequence. Each curve has a similar slope, but the sequence of zeroes is sorted much faster, likely because there is only one digit. In this case, the equation O(d(n+N)) becomes O(n+10) = O(n), and is essentially linear. This represents the fastest possible input for radix sort, although no case will affect its asymptotic complexity of O(d(n+N)). Figure 3F plots radix sort's size versus its running time divided by its expected time when operating on a random sequence. Since my tests bounded the randomly generated integers by the size of the sequence, the number of digits varies between 1 and 4 depending on the input sequence size. I adjusted for this in calculating the expected time for different input sizes. For example, individual integer elements used in testing the sequence with a size n = 128 could be 127 at the greatest. 127 has 3 digits, or dimensions. Therefore, since the size n is 128, the number of dimensions d equals 3, and the number of buckets N is 10 for the digits 0 through 9, the expected time for this input is O(3(128+10). Similar adjustments for each input size result in a linear path after n = 64. This suggests the Big-O pair (64, 3.88E-08).

Comparison of Results

Figure 1 shows all six sorting algorithms on a log-log plot. Sequences of randomly generated integers were used as input for all sorts. This provides an average case for each sort and allows for accurate comparison between the sorting algorithms.

As expected, radix sort is the most efficient. This sort is ideal for integer values and its curve runs well beneath those of the comparison sorts.

The variation between merge sort and heap sort is interesting because these algorithms share a complexity of O(nlogn) for all cases. Heap sort was faster for all sizes tested. However, it is important to note that the range of tested inputs does not include large sizes of the scale of those used to perform certain tasks on more powerful systems. In this experiment I suspect that the relative speed of heap sort is due to the O(n) comparisons of each combination step of merge sort. Heap sort involves only as many comparisons as are necessary to maintain heap order.

For small and medium-sized inputs, the quick sort with random pivot selection is faster than both merge sort and heap sort, although its curve eventually falls in line with that of heap sort. That it is faster is notable because quick sort is not guaranteed an efficiency of O(nlogn); this is only expected. However, quick sort is observed to outperform consistent O(nlogn) algorithms, such as merge sort, which is guaranteed to split a data set into two even subsequences at each step. By efficiently performing the majority of work at the division step with n-1 comparisons, rather than the merge step with O(n) comparisons, quick sort may have an advantage. Along with the relatively smooth slope of quick sort's curve, that quick sort surpasses the speed of the merge and heap sort algorithms proves that its randomization is reliable.

The two priority queue sorting algorithms with quadratic complexities, insertion and selection sort, appear competitive with the others for small input sequences. However, beyond the size n = 64 they betray their quadratic complexities. Were their paths to continue, it is clear they would distance themselves even further from the more efficient algorithms. This is as expected; the average O(n^2) complexity is efficient only at very small input sizes. At the largest test size, n = 16384, the high cost of these two sorts forced the test program to terminate. This proves their inefficiencies more than any graph could. Comparing the two, insertion sort is consistently faster. This also confirms analysis: selection sort always makes the maximum number of comparisons, while insertion sort performs as many comparisons as necessary. When sorting a randomly scrambled sequence, the required comparisons will be less than the worst case or maximum number of comparisons involved in an ordered sequence. As a result the sorting times are plainly distinguished.

Summary

This report compares the results of efficiency experiments of sorting algorithms to their expected outcomes, as determined by theoretical analysis. Data resulting from the experiments confirms the asymptotic complexities of the sorts over a range of inputs. Plots of the experimental data provide insight into the interrelations and preferred uses of the various methods of sorting.

Radix sort is proven to be extremely and consistently fast, while insertion and selection sorts are demonstrated to have quadratic complexities that are undesirable for all but small input sizes. An interesting comparison of the linearithmic algorithms, merge sort and heap sort, suggests that heap sort can be much more efficient, at least for moderately sized sequences. Lastly, an analysis of various quick sort implementations provides an understanding of the uses of various methods of pivot-selection. While all strategies output similar results, randomly selecting a pivot proved faster on average over the range of input sizes. Using this method, quick sort was faster than all sorts except for radix sort. This emphasized what superficially may seem an illogical concept: the value of randomization in sorting.

I feel these experiments and the analysis of their results has entrenched a thorough basic understanding of sorting algorithms. Perhaps most importantly, this report has demonstrated that no sorting method is ideal for every scenario. The positive and negative aspects of these algorithms must be considered, but first understood, before one can best navigate a sorting problem.

Plots

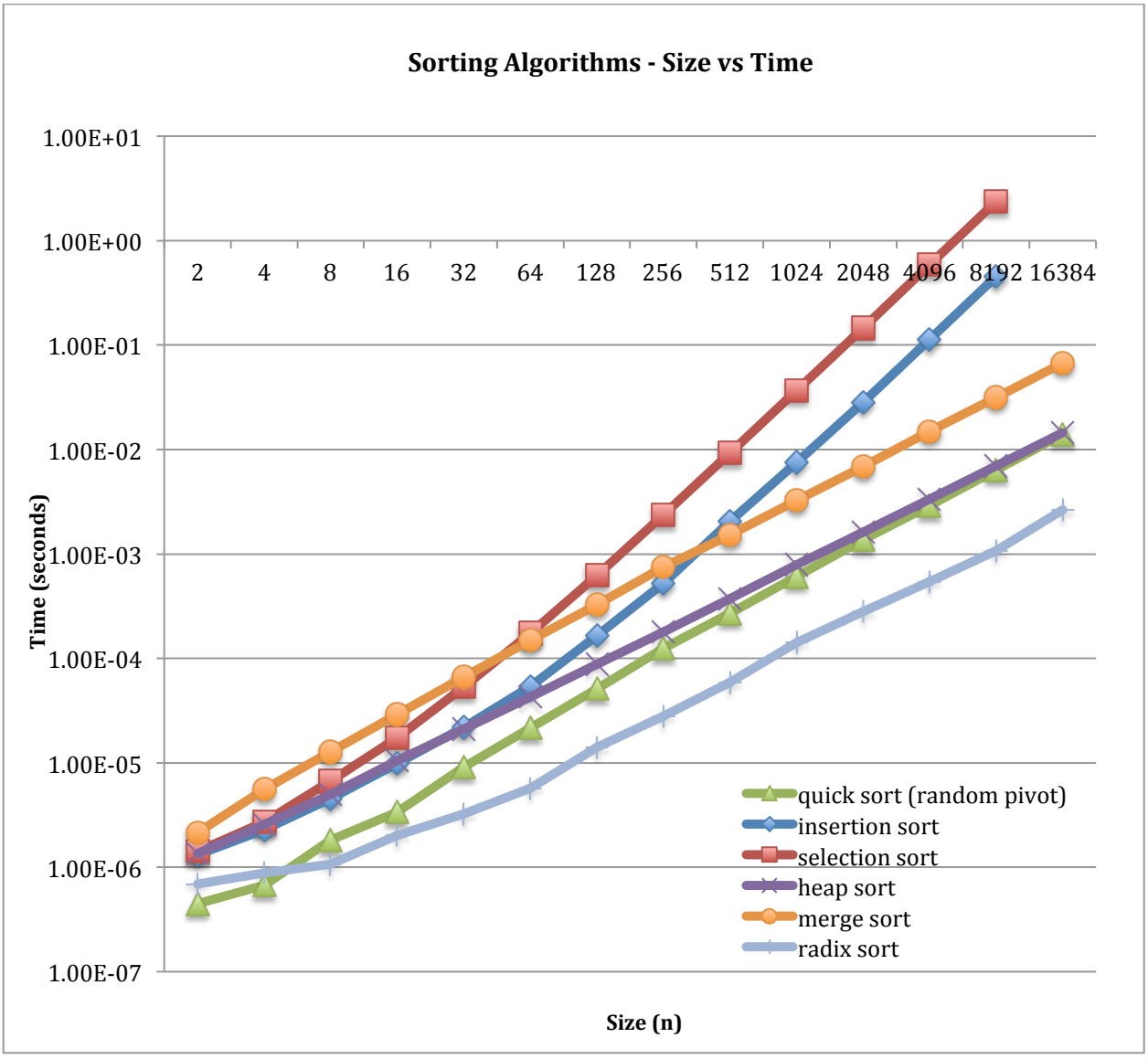

Figure 1: Execution time in seconds vs. sequence size for all sorting algorithms are shown on a log-log plot. Random input sequences were used to provide fair comparison. The quick sort implementation in which pivots are randomly selected is used, as it proved more efficient for random inputs than the other quick sort implementations. Time values for the two quadratic priority queue sorts, selection sort and insertion sort, could not be determined for the maximum size of 16384 due to the long running time and limited computer capacity; however, their quadratic paths are clearly evident.

Sorting Algorithm Plots: Time vs. Size

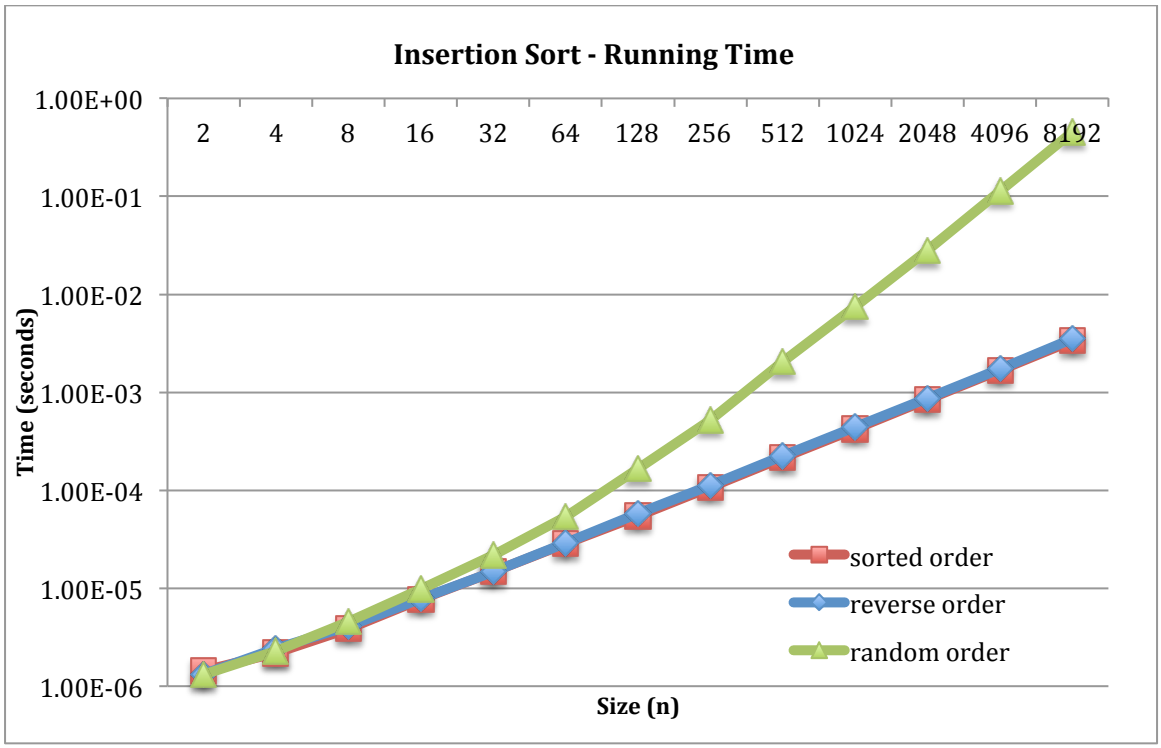

Figure 2A: Execution time vs. size for insertion sort on a log-log plot. Best case is represented by the reverse-sorted sequence. Worst case is represented by the sorted sequence. Average case is represented by a randomized sequence.

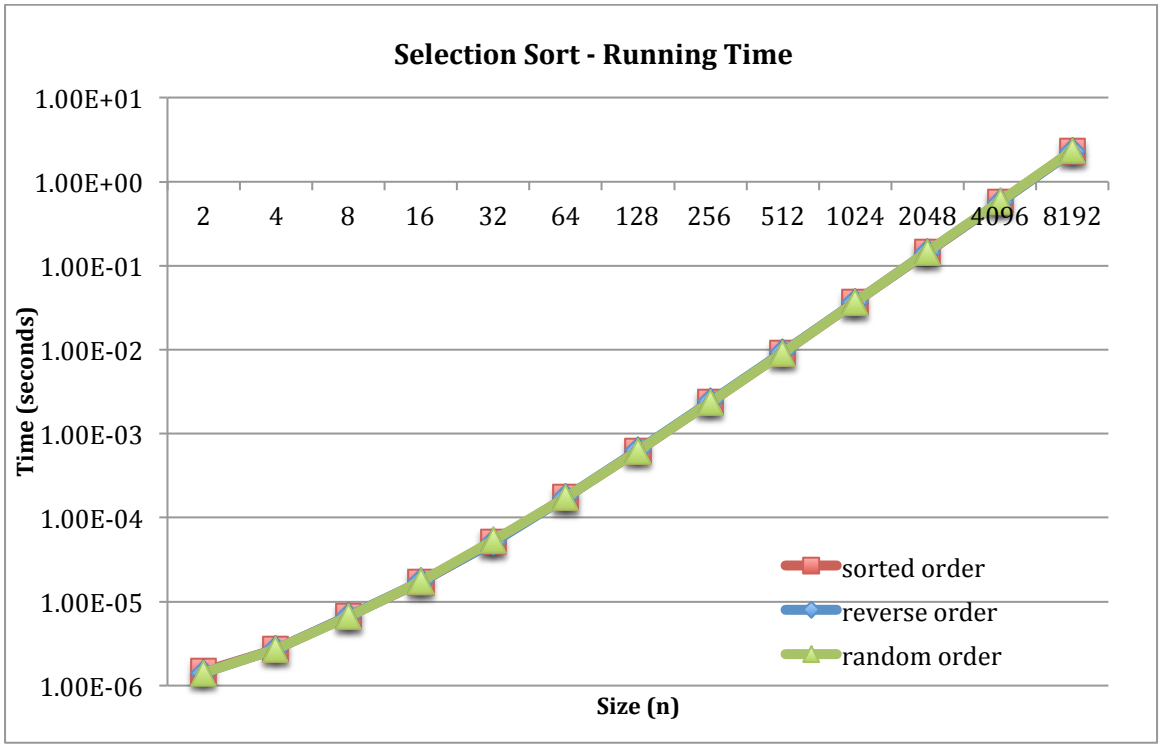

Figure 2B: Execution time vs. size for selection sort on a log-log plot. Sorted, reverse sorted, and randomized input sequences are depicted.

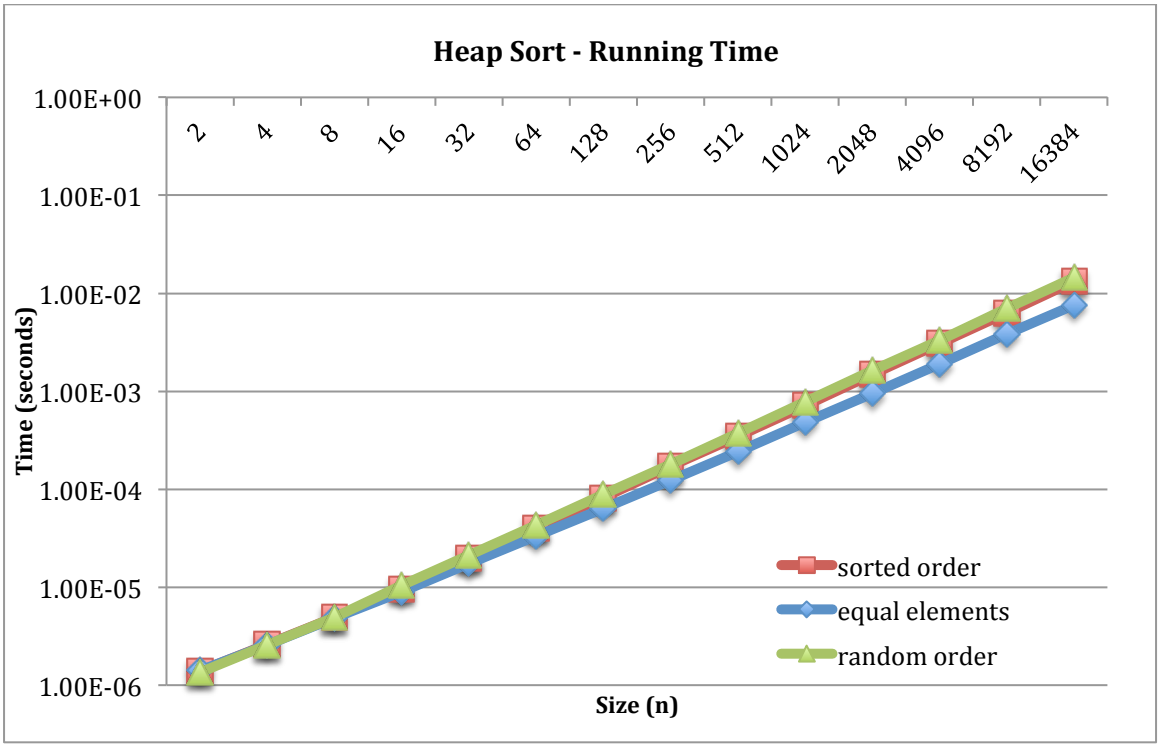

Figure 2C: Execution time vs. size for heap sort on a log-log plot. A sorted sequence, a sequence of zeroes, and a randomly generated sequence are used.

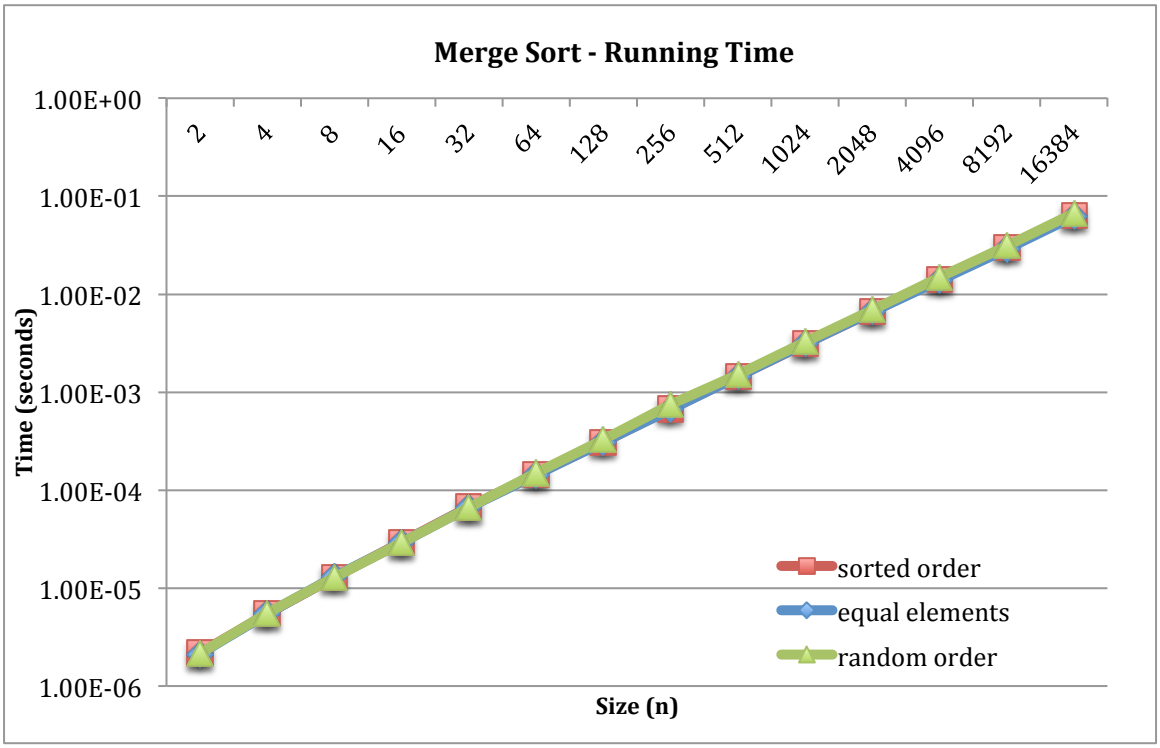

Figure 2D: Execution time vs. size for merge sort on a log-log plot. A sorted sequence, a sequence of zeroes, and a randomly generated sequence are used.

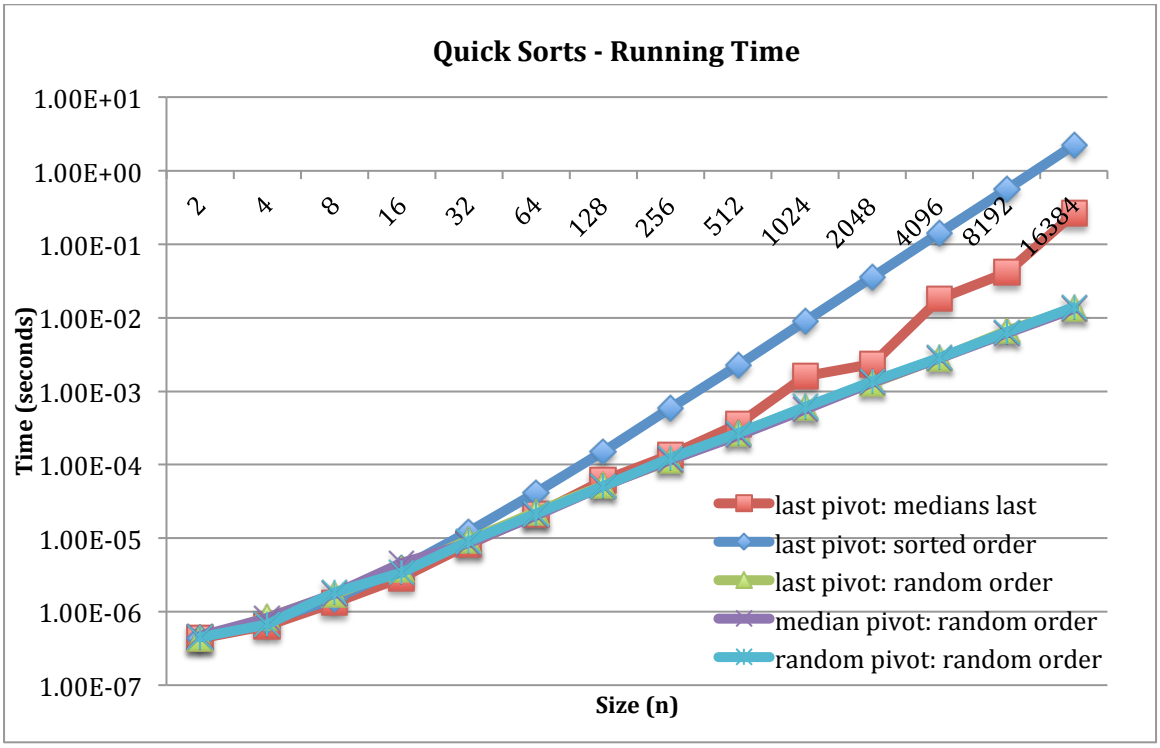

Figure 2E: Execution time vs. size for quick sort implementations on a log-log plot. Three pivot-selecting methods are shown: last, median, and random. For the last element pivot selection implementation, sorted order represents the worst case, and a special sequence with each subsequence's median at the end of it is intended to represent the best case. For the other implementations only the plots of a random input sequence are shown.

Figure 2F: Execution time vs. size for radix sort on a log-log plot. A sorted sequence, a sequence of zeroes, and a randomly generated sequence are used.

Priority Queue Sorts: Big-O Constants

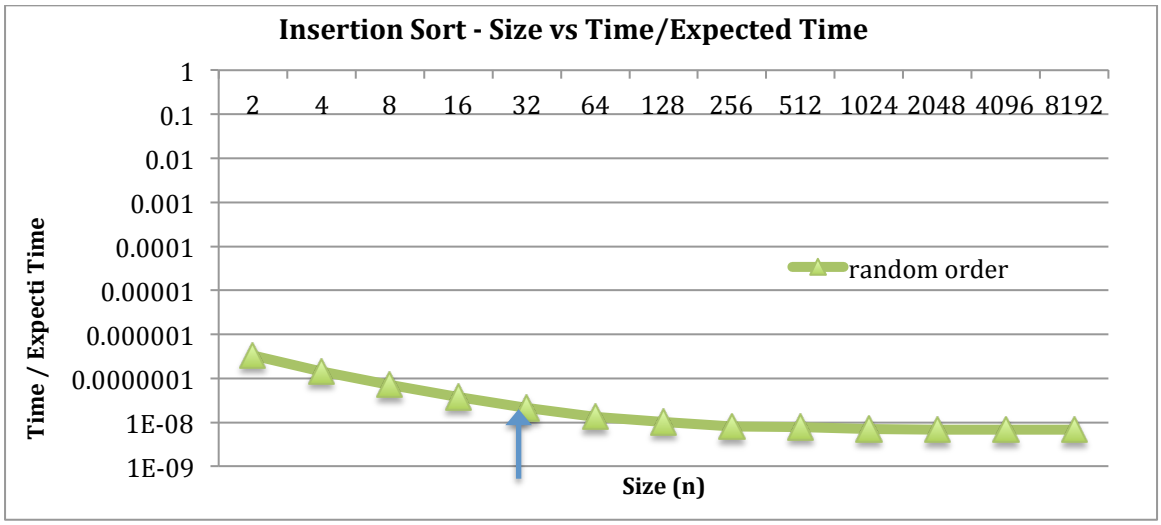

Figure 3A: Execution size vs. time/expected time (n^2) for insertion sort on a random input sequence. An arrow marks the location of the determined Big-O constants (k0, C).

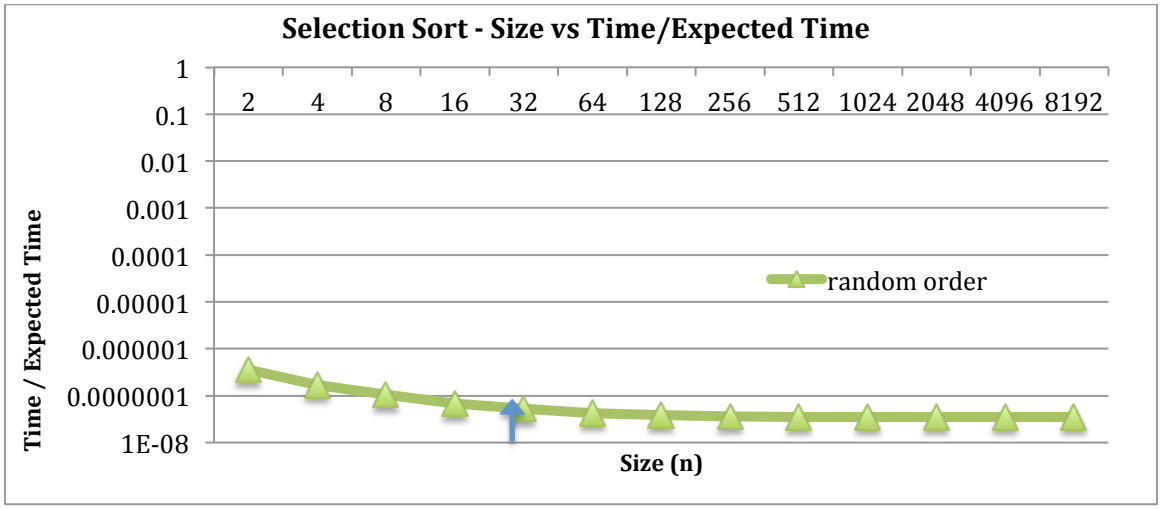

Figure 3B: Execution size vs. time/expected time (n^2) for selection sort on a random input sequence. An arrow marks the location of the determined Big-O constants (k0, C).

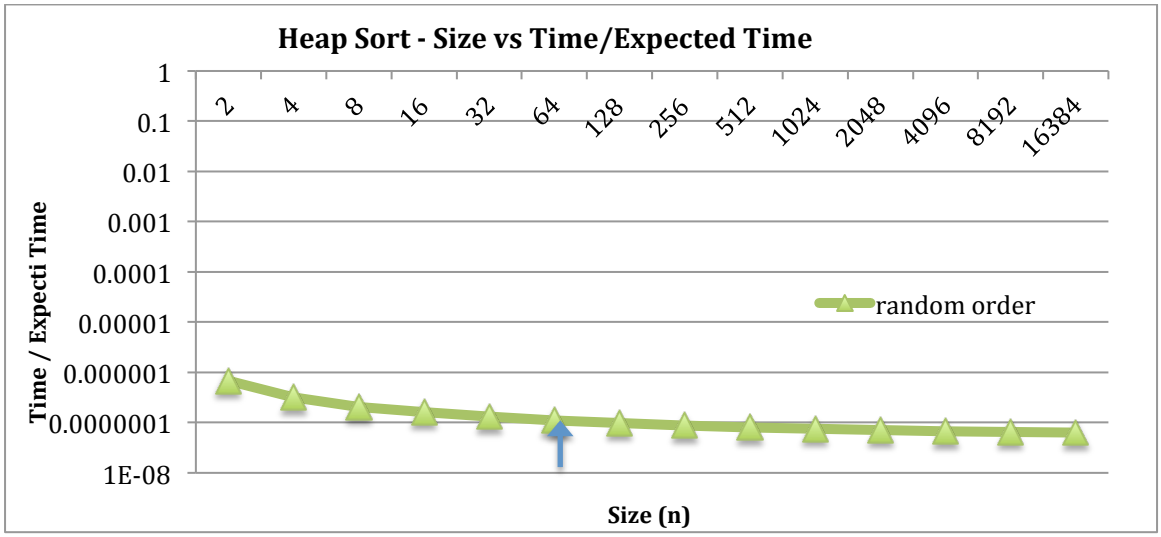

Figure 3C: Execution size vs. time/expected time (nlogn) for heap sort on a random input sequence. An arrow marks the location of the determined Big-O constants (k0, C).

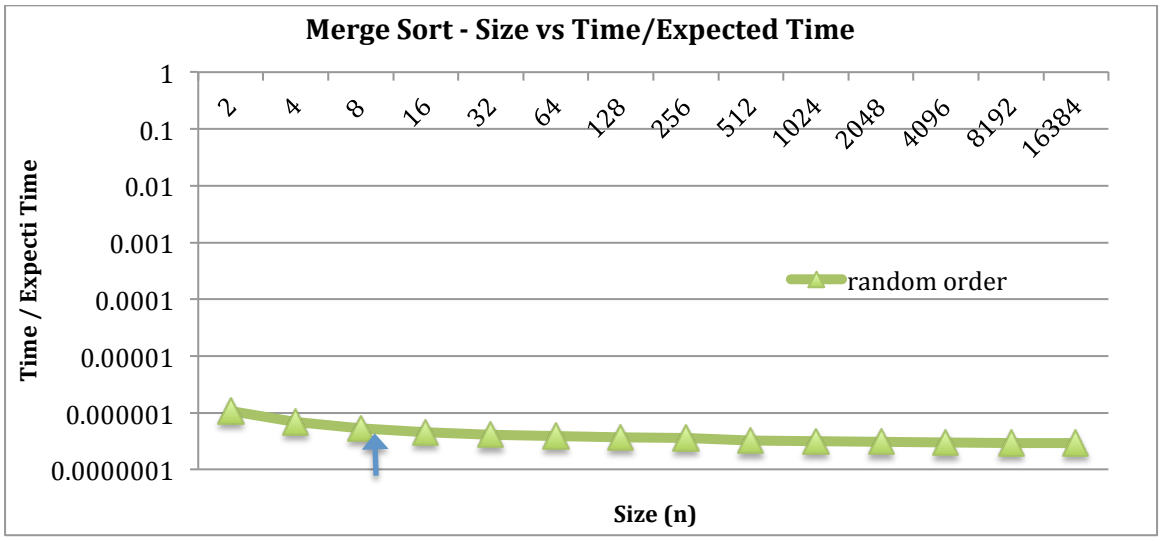

Figure 3D: Execution size vs. time/expected time (nlogn) for merge sort on a random input sequence. An arrow marks the location of the determined Big-O constants (k0, C).

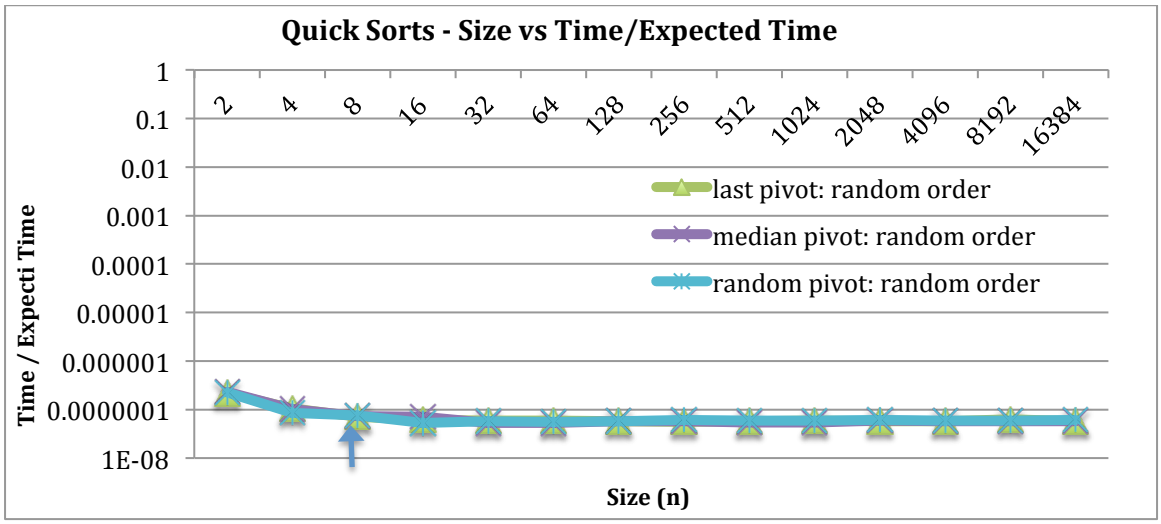

Figure 3E: Execution size vs. time/expected time (nlogn) for quick sorts on random input sequences. An arrow marks the location of the determined Big-O constants (k0, C).

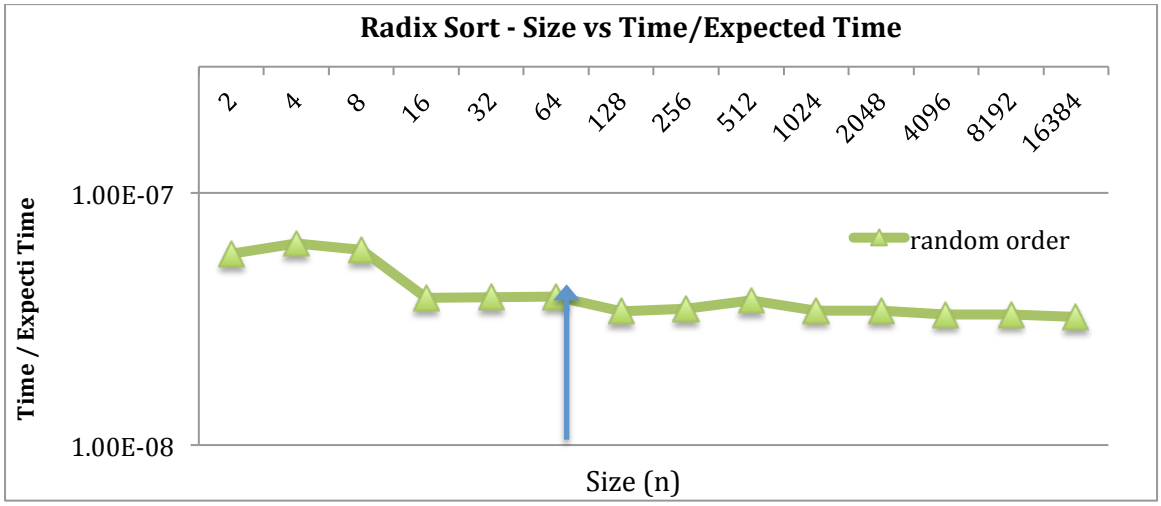

Figure 3F: Execution size vs. time/expected time (n) for radix sort on a random input sequence. An arrow marks the location of the determined Big-O constants (k0, C).