Robert Vo
CECS 619 - Introduction to Algorithms
Spring 2017 Semester
Project 1
Due Date: 3/21/2017

Overview

The purpose of this project was to statistically analyze the efficiency of five sorting algorithms: insertion sort, merge sort, quick sort, heap sort, and radix sort. A template was provided in order to emphasize on the analysis of the algorithm, and not on the coding. Tests were conducted with various input values to determine whether run time followed asymptotic time complexity of each sorting algorithm. Furthermore, the comparison sorts of insertion, merge, heap, and quicksort were analyzed even further to determine whether the number of comparisons were good predictors for the run time. Numerous plots are provided in the project and description of each section describes the details of the plots and any errors that may have appeared in the experiments.

I. Theoretical Analysis

Insertion Sort: The process of insertion sort is incremental. This sort simply selects the next element in order, compares it with the previous element, and insert if necessary. It is a comparison sort that traverses the entire length of the list.

- Best case: Input is already sorted. The algorithm will run in linear time because the number of comparisons will be n-1 comparisons. Therefore, the number of comparisons will simply traverse the entire input.
  Time Complexity O(n)

- Average case: Random order of input. It is expected that half of the elements are less than the element to be inserted. Therefore, the average insertion will cause 1/2 of numbers in the input to shift with each insertion.
  Time Complexity O(n^2)

- Worst case: Input is in reverse order. The number of comparisons is linear and each element inserted will shift the entire list.
  Time Complexity O(n^2)

Merge Sort: Merge sort is independent on the order of the input. This sort operates with the Divide and Conquer Paradigm and therefore, all cases run times are the same for best, average, and worst. The combining step of the sorted sublists does O(n) work, and the dividing step runs at for the height of the tree, log n levels with n number of leaves. Therefore, all cases are the same.

- Best case: *Time Complexity O(nlogn)*

- Average case: *Time Complexity O(nlogn)*

- Worst case: *Time Complexity O(nlogn)*

Quick Sort: Quick sort is determined by the partitioning done by the pivot element. The pivot element can be any value in the array, and the chosen value determines how the remaining array is partitioned. Balanced partitions will result in the best case scenarios, and unbalanced partitions can result in the best case also. However, extremely unbalanced partitions where one partition has zero elements and the other partition has n-1 elements results in the worst case. Average cases of quicksort are due to alternating between random values that may choose lucky vs. unlucky partitionings. The result will still be similar to its best case.

- Best case: Input in order and the pivot chosen is the median which results in a constant time and even partitioning. In addition, input partitioned evenly into two n/2, or by known fraction (i.e. 9/10 & 1/10 or 1/100 & 99/100) will run in best case. It will run similar to merge sort. When already sorted, the median can be found in constant time.
  Time Complexity O(nlogn)

- Average case: A randomized input will result in an average case. The alternating between lucky and unlucky cases will still result in a time complexity equivalent to the best case. For instance, one subproblem can result in a balanced split, followed by the next subproblem with an extremely unbalanced split. At the end, this will still result in a best case complexity.
  Time Complexity O(nlogn)

- Worst case: Input in increasing or decreasing order with the pivot being the first or last element. Bad partitioning. If one partition has 0 elements, and another partition has n-1 elements., or any partition from 0, n-1; 1, n-2; 2, n-3……..n-2, 1; n-1, 0. This is the most unbalanced a split can be, and will therefore result in the worst case. The worst case occurs due to the pivot being chosen as the minimum or maximum element.
  Time Complexity O(n^2)

Heap Sort: Heap Sort is an in-place algorithm that uses a heap data structure. Input of random, increasing, or decreasing order will follow the heap data structure, and therefore, all cases are the same complexity. The max-heap data structure contains the maximum element at the root position. Heap Sort involves swapping that root element with the last element in the structure, and removing it. Therefore, all cases for heap sort are determined by the height of the tree.

- Best case: *Time Complexity O(nlogn)*

- Average case: *Time Complexity O(nlogn)*

- Worst case: *Time Complexity O(nlogn)*

Radix Sort: Radix sort is the only sorting algorithm that is not based off comparisons. It uses the counting sort algorithm and divides the number or word to determine the appropriate number of passes and size of the division. Input type does not have great effect on the running time, but the length of the number or word does. It is a stable sort that maintains the order of elements. The least significant digit is chosen and sorted, and the algorithm progresses through each digit. Therefore, complexity is based on the how the elements are divided and the number of passes required.

- Best case: *Time Complexity O(d(n + k))*
- Average case: *Time Complexity O(d(n + k))*
- Worst case: *Time Complexity O(d(n + k))*

II. Data Generation and Experimental Setup

The data chosen for this project was determined by the run time. Run times were chosen to be between 5000-15000 ms for most cases. However, there were exceptions in order to plot on the same graph. For example, Figure 4B shows the best cases for quicksort, merge, heap, and radix with very large input value. It can be seen from the plot that merge sort's maximum run time was only 6000ms. This was done in order to provide one plot for better comparison with the other sorts. Inputs ranged from tens of thousands to hundreds of millions. In the case of hundreds of millions, insertion sort's best case is linear. Therefore, it handled a large number of elements. In contrast, its worst case is n^2 so therefore, values were greatly reduced to tens of thousands.

Inputs also varied in the type of input such as increasing, random, and decreasing. They were chosen based on the sorting algorithm, and remained consistent. For example, testing the worst cases for merge, heap, and radix sorts, the input type was maintained for all of those with an increasing order type, as well as input size. Also, certain algorithms needed certain input to perform best case. For instance, the quicksort algorithm with a median input functioned better with an input type of increasing order. Furthermore, the quicksort algorithm with first element as pivot greatly varied in performance depending on the input type. Each experiment was run a minimum of three times, and a few cases 5 times if plots did not show a good result i.e. inconsistent, sharp changes, etc.

The machine used for the project was my personal laptop:

Specifications
Model - MacBook Air
Processor - 1.6 GHz Intel Core i5
Memory - 8 GB 1600 MHz DDR3
Current OS - macOS Sierra Version 10.12.3

Timing Mechanism
The timing mechanism used for my machine is provided by the Standard C Library. The clock() function determines and returns the processor time used upon the call of a process. The time value is measured in CLOCKS_PER_SECs of a second. The recording of the run times executed were in milliseconds. However, the algorithm provided returned the execution time in microseconds and seconds.

III. Quicksort Analysis

The performances of quicksort vary depending on the input as well as the pivot. A chosen pivot that provides a fractional split will always result in the best case performance i.e. 9/10:1/10 split; 1/2:1/2 split, etc. The best cases for the three quicksorts are provided in the plot below this text. From Figure 3A, it shows that the quicksort with a median pivot performed the best/fastest with this large input. However, it is important to note that all of the quicksorts are performing with a best case time complexity of O(nlogn) with slight differences caused by constant values.

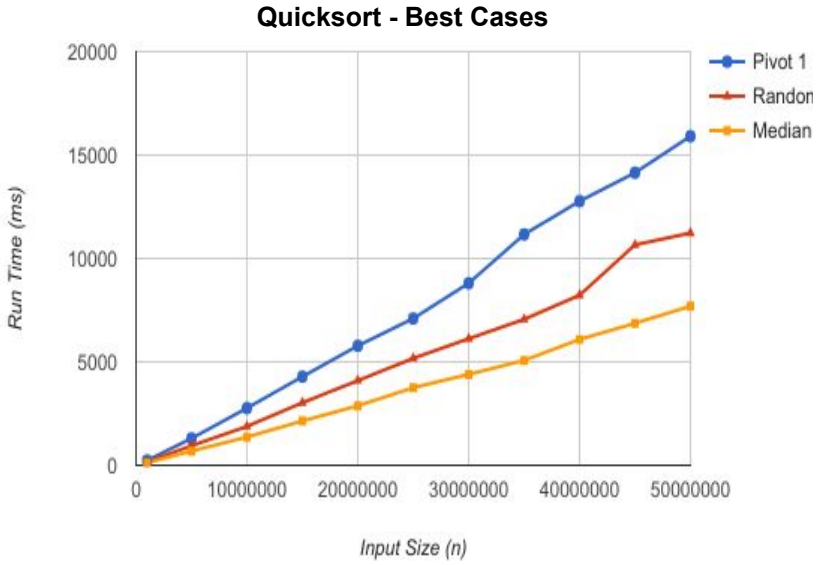


Figure 3A

The average case was performed with random input for all cases. The average case occurs due to partitioning not always happening to be balanced. A balanced partition occurs with a 1/2:1/2 split by a median pivot. However, unbalanced partitioning may occur due to a pivot splitting the partitions in a 3/4:1/4 split, for example. Other cases may result in zero splits and may cause one partition to have zero elements, while the other partition has n-1 element (resulting in the worst case). The average case occurs due to randomization of a chosen pivot that can cause an alternation between worst case and best case partitioning. Even though slight variances in the constant factor will result in different running times, the constant will be absorbed in the big-O notation. Thus, time complexity for average case will be O(nlogn). Figure 3B shows that the random input greatly affects the quicksort algorithms.

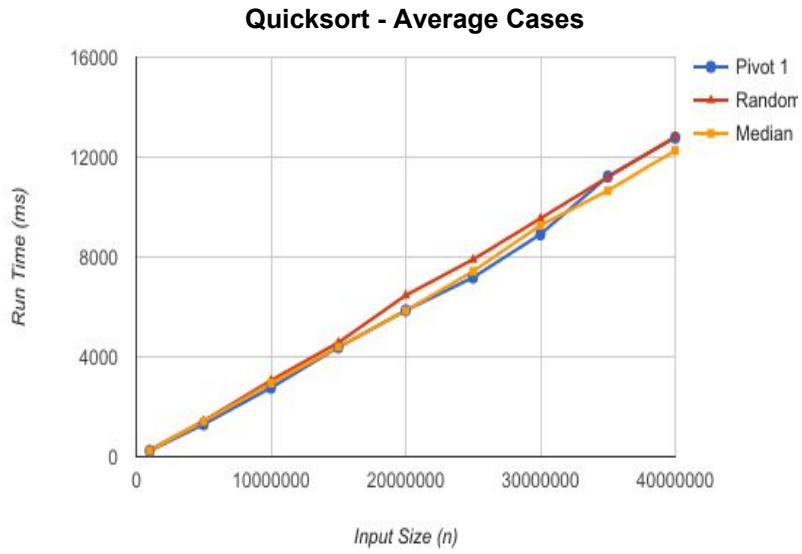


Figure 3B

The worst case scenario for quicksort occurs when a split results in the most unbalanced partitions. That is when the pivot is chosen as the lowest or highest value, which will cause a partition of zero elements for one side, and n-1 elements for the other side. For the two plots below, it shows that the random pivot quicksort performs the best worst case, and the quicksort that chooses the the first element as pivot will perform the worst. The reason for this worst case scenario is due to the input being in increasing order. Thus, pivot 1 will always choose the first and lowest element in the list causing extremely unbalanced partitions. Comparing Figures 3C and 3D, 3C shows the run times for worst case inputs of Pivot 1 and Median with input size much less than the random input. Therefore, Figure 3C shows that the worst algorithm amongst the quicksorts is the Pivot 1.

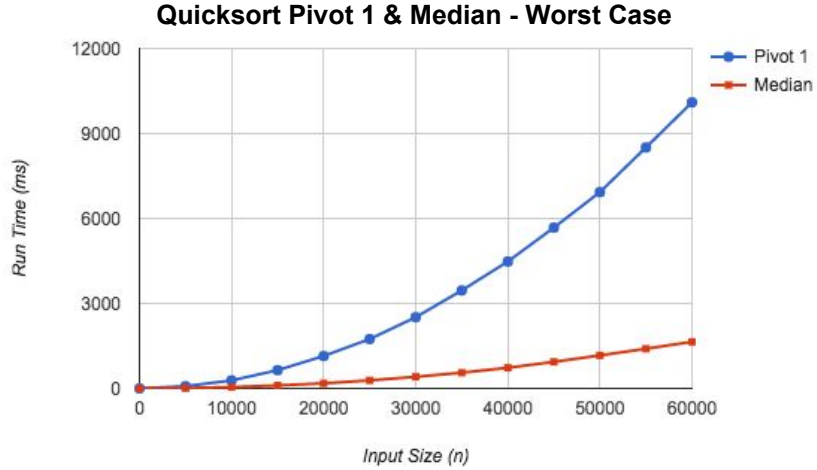


Figure 3C

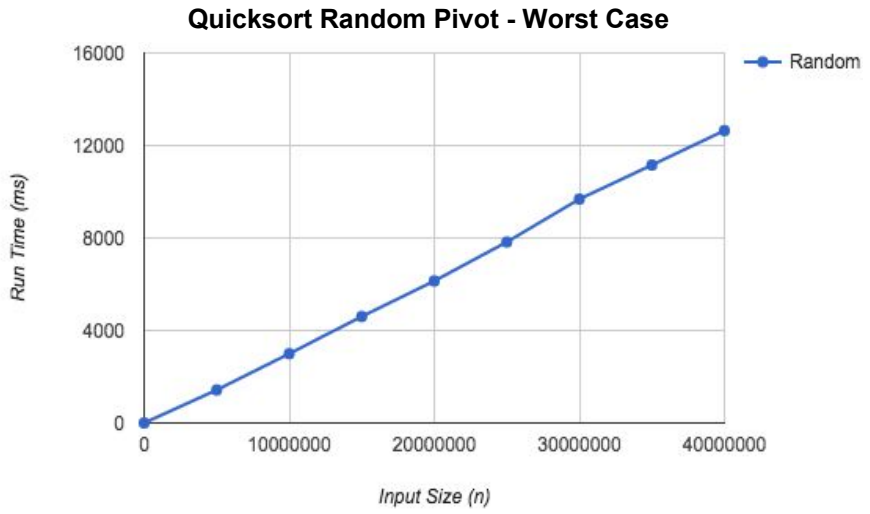


Figure 3D

IV. Sorting Performances

Best Case

The best case running times for the sorts are seen below. Figure 4A shows the best case for insertion sort which runs at O(n). For that reason, the input values in increasing value for insertion sort are extremely large (hundred millions). Since insertion sort performs in linear time, it is the best with extremely large values. However, this is when the list is already in sorted or almost sorted order.

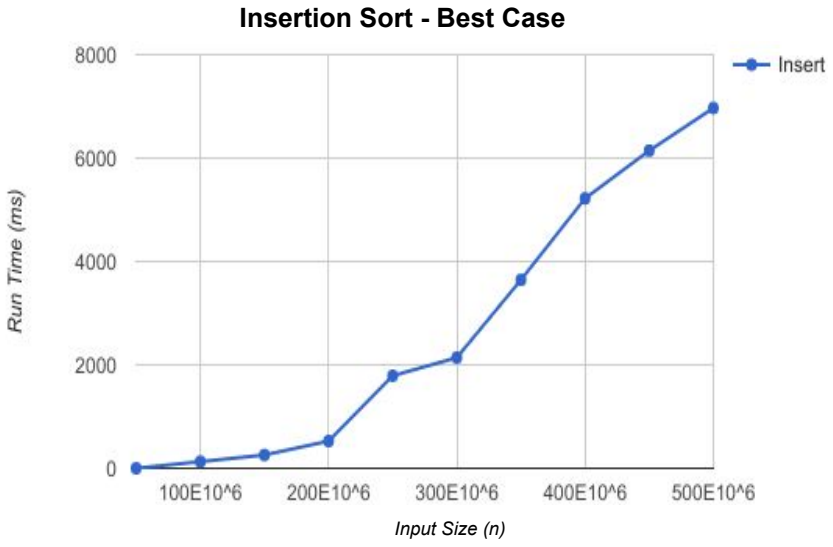


Figure 4A

The remaining sorts, quicksort, merge, heap, and radix are plotted on a separate graph because of lower input values in increasing order. Figure 4B shows the best performance out of all of them as merge sort.

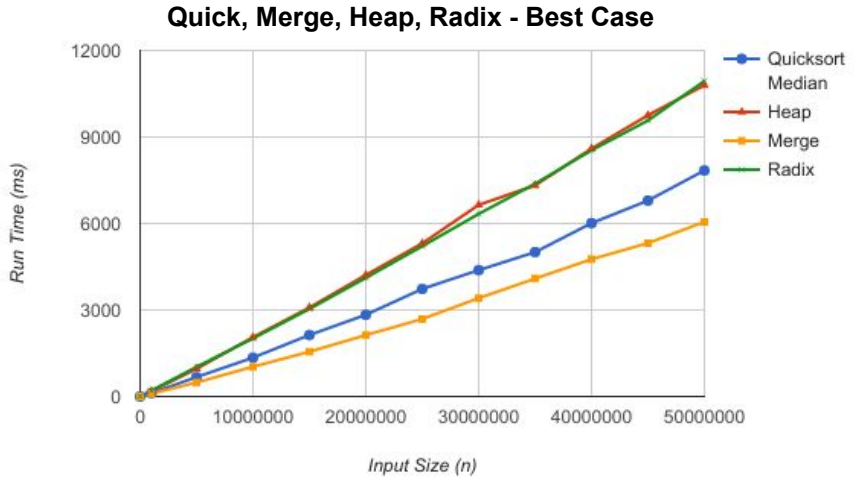


Figure 4B

Worst Case

The worst case scenarios are displayed in two graphs. Figure 4C shows the main difference between insertion & quicksort vs. radix, merge, and heap sort. It can be seen from the plot that only quicksort and insertion sort are obviously displayed, while heap, merge, and radix run along the x-axis. Since the worst case time complexity for insertion and quicksort is n^2, it was more suitable to put these two sorts on their own graph with much smaller input size. This plot shows that at worst case, insertion sort performed better than quicksort. However, both of these sorts are extremely slow when it comes to their worst case.

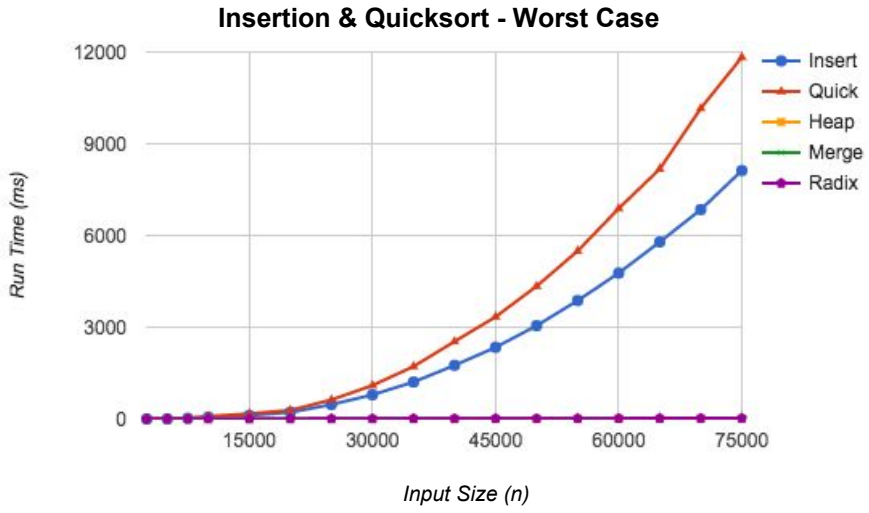


Figure 4C

This next plot shows the worst case for heap, merge, and radix sort. The worst case for these sorts are the same as their best and average cases, which explains how they still perform well with an input in the millions. Figure 4D shows that at worst case, merge sort performed the best. And overall, performed the best for all sorts.

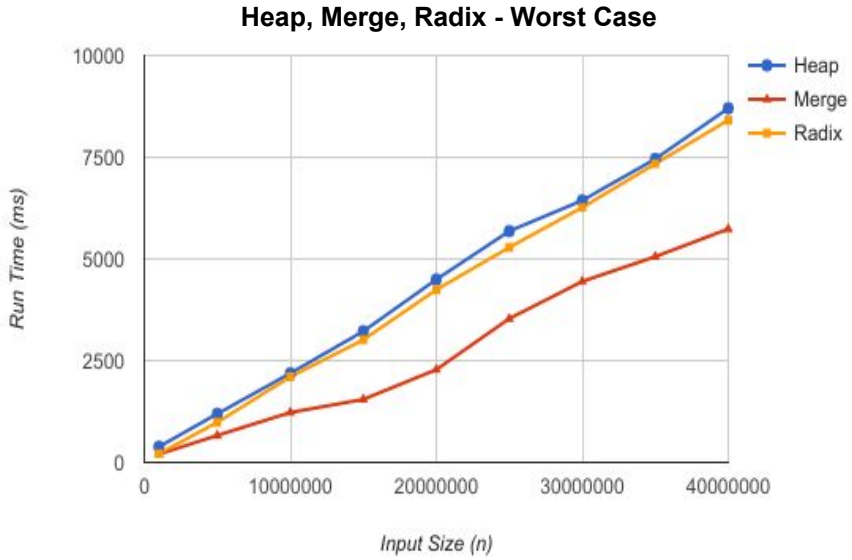


Figure 4D

Average Case

The average cases are also plotted on two distinct plots. Figure 4E shows that the average case for insertion sort still performs at time complexity $O(n^2)$, similar to its worst case. Thus, input value is extremely low.

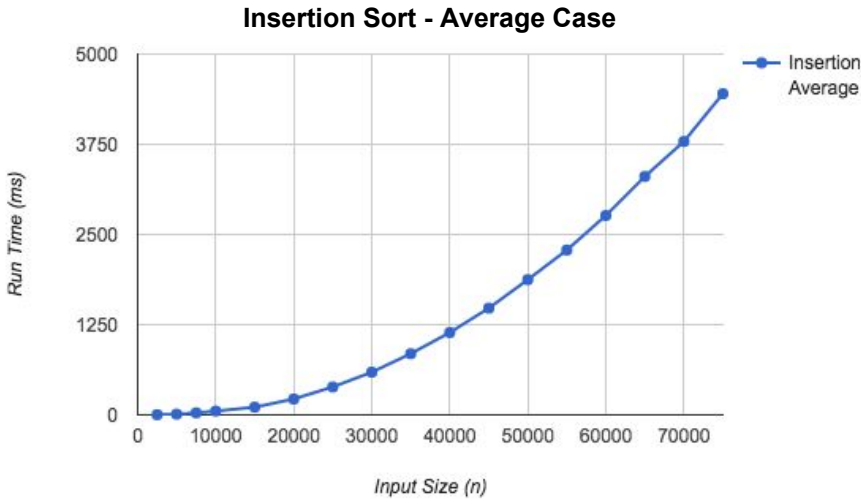


Figure 4E

The average case for quicksort, heap, merge, and radix in Figure 4F shows that merge sort performed the best like in worst case scenario, followed by quicksort. This showed that the two divide and conquer algorithms performed the best on average.

Figure 4F

The results of part IV show us that out of the five sorts, merge sort and quicksort perform the best/fastest. Since we are mainly concerned with asymptotic analysis, the worst case scenario gives the best representation for performance. Merge sort and quicksort are divide and conquer algorithms, which gives a good indication as to how they perform well in O(nlogn) time.

V. Time/Complexity

This part of the project was to determine whether the best, average, and worst case theoretical analyses agreed with the experimental results. In order to determine if that was the case, each sort was plotted as (run time/complexity) vs. input size. Since we are only concerned with asymptotic analysis, the input size has to be greater than a certain point in order to see the noticeable difference. That point is n_0. Therefore, $n>n_0$.

The majority of plots displayed relatively straight lines with respect to how the plot was created. Therefore, the experiment results agree with the theoretical asymptotic analyses. The plots, since using milliseconds, resulted in extremely low values for the y-axis. Completely straight lines were not possible due to the graph being “zoomed in.” If graphs were plotted with larger or integer values, straight lines would be much more noticeable without major spikes or dips. Also, graphs demonstrate strong changes at the beginning due to the plot being asymptotic. This means that the effects of input size will not be affected unless the plot shows a value of $n > n_0$.

Figures 5A - 5C show extremely straight lines for the cases of insertion sort.

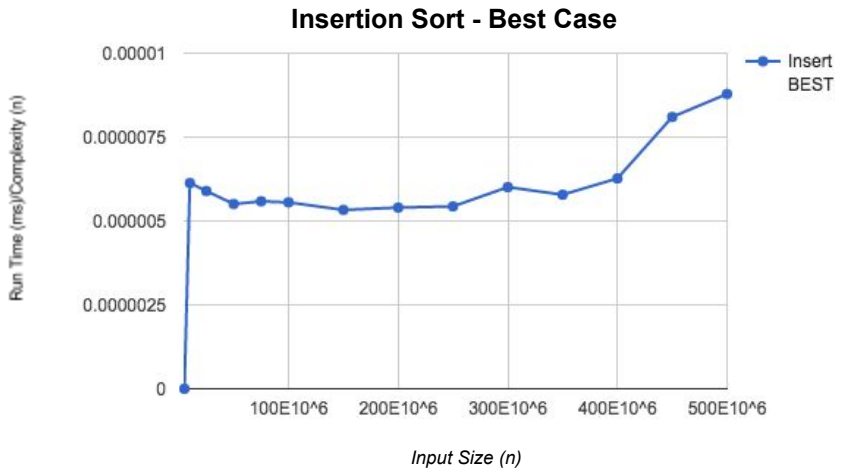


Figure 5A

Figure 5B

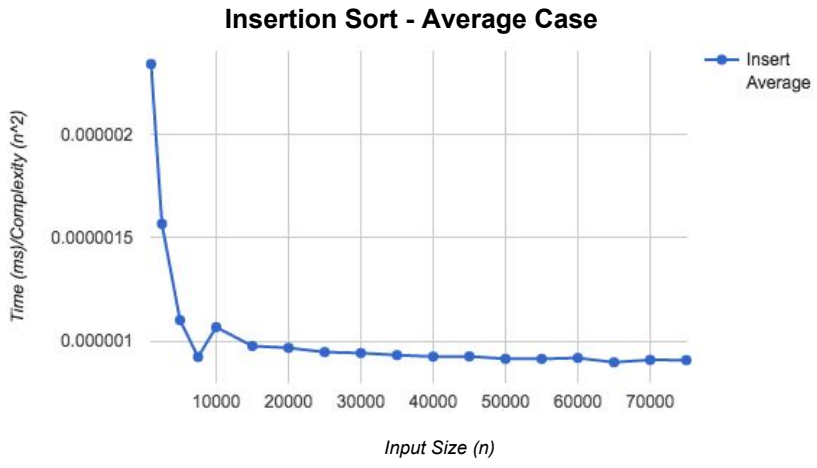


Figure 5C

Figures 5D - 5F show the experimental results for merge sort. Best and average cases show relatively straight lines. However, the results for worst case show a strong dip midway through the plot. This reasoning may be due to computer performance.

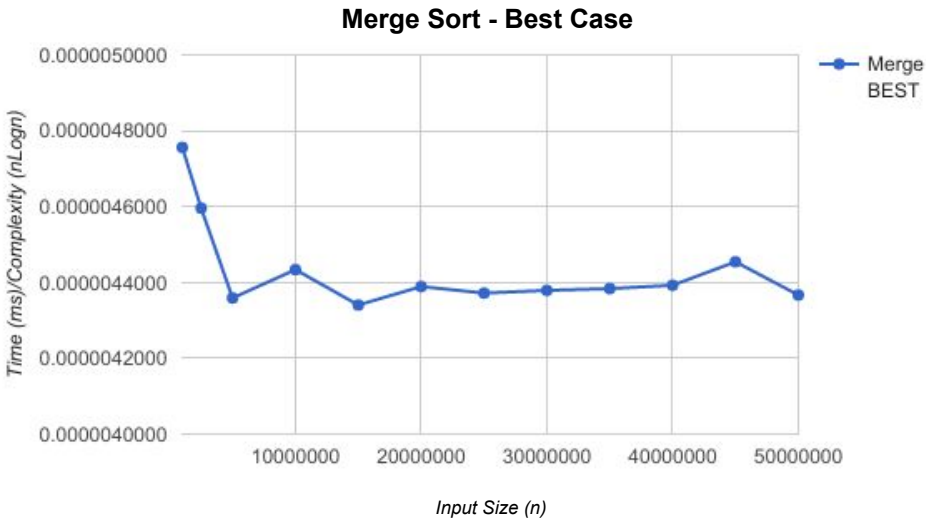


Figure 5D

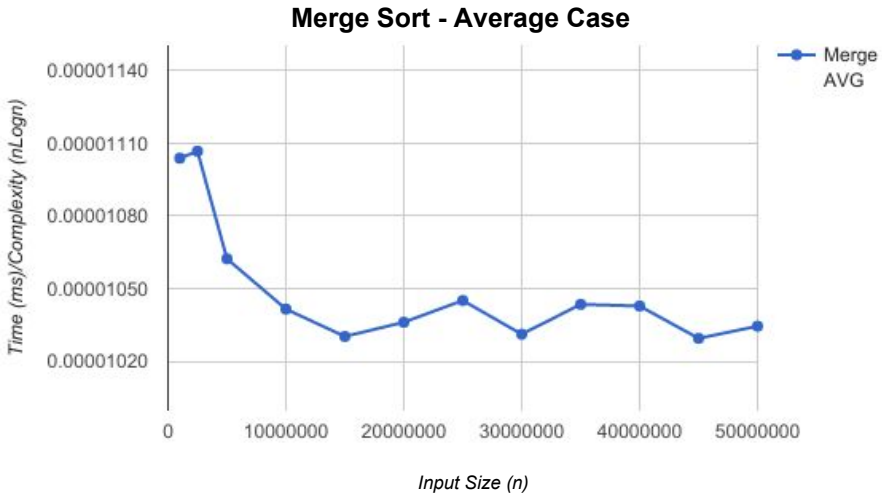


Figure 5E

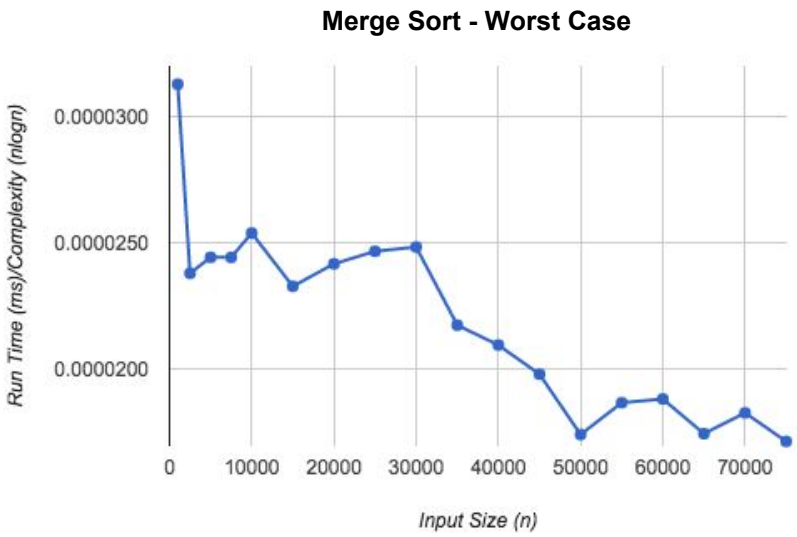


Figure 5F

Figures 5G - 5J show the results of heap sort. Other than small fluctuations, the graphs maintain steady lines throughout.

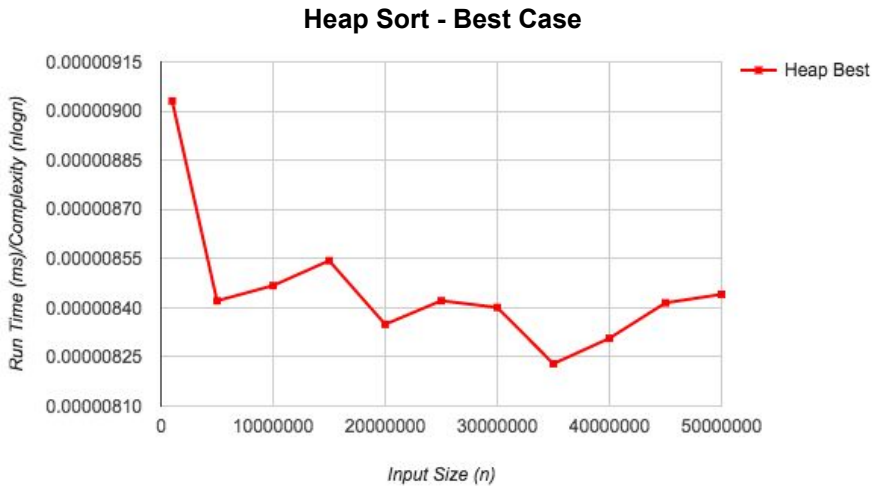


Figure 5G

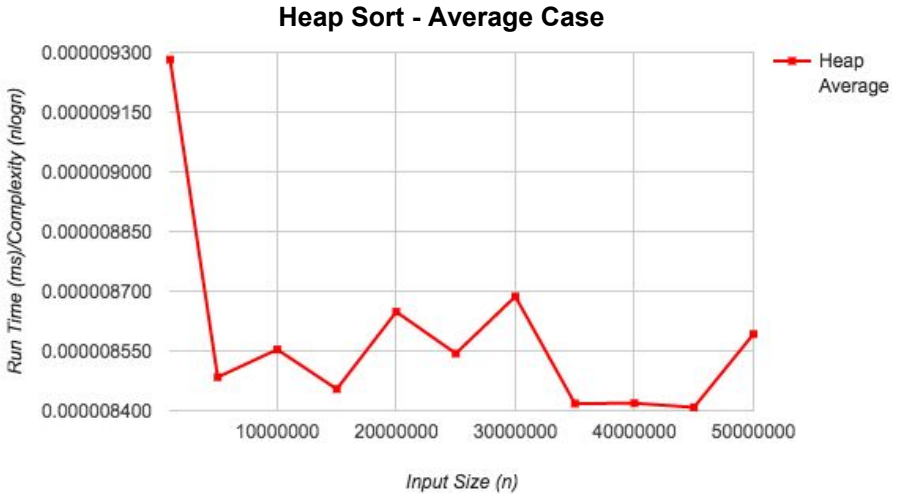


Figure 5H

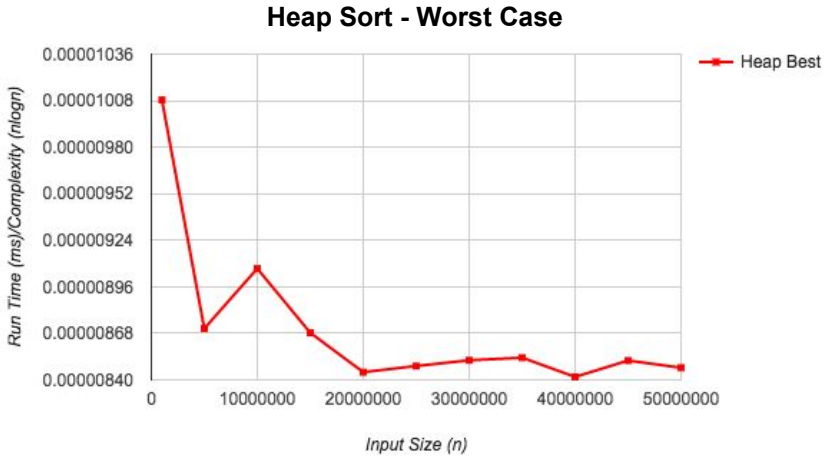


Figure 5J

Figures 5K - 5M show radix sort. Radix sort showed the greatest difference between all the sorts. Regardless, it showed consistency with all cases; A strong rise at the beginning is consistent throughout, followed by a leveling out once the input size increases.

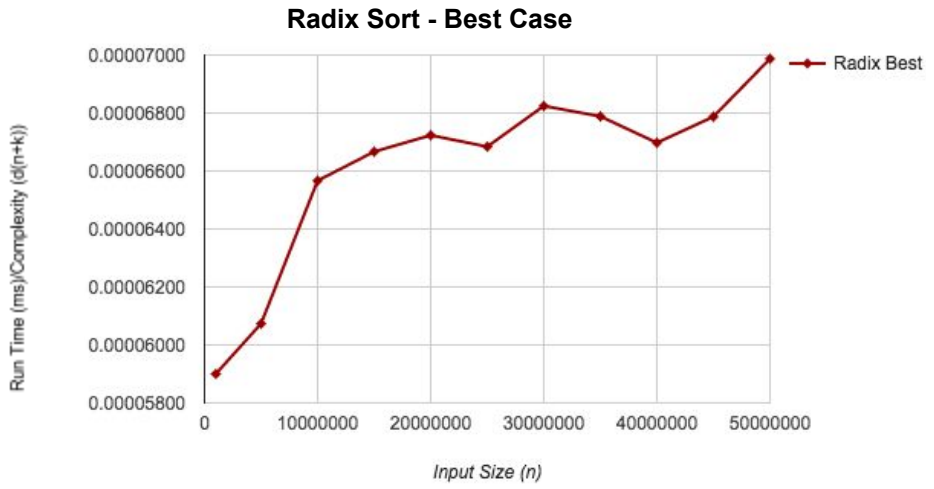


Figure 5K

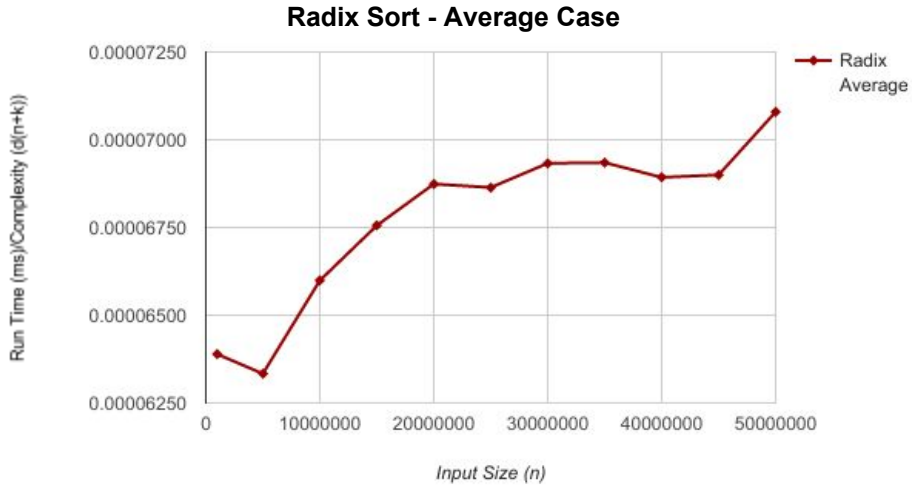


Figure 5L

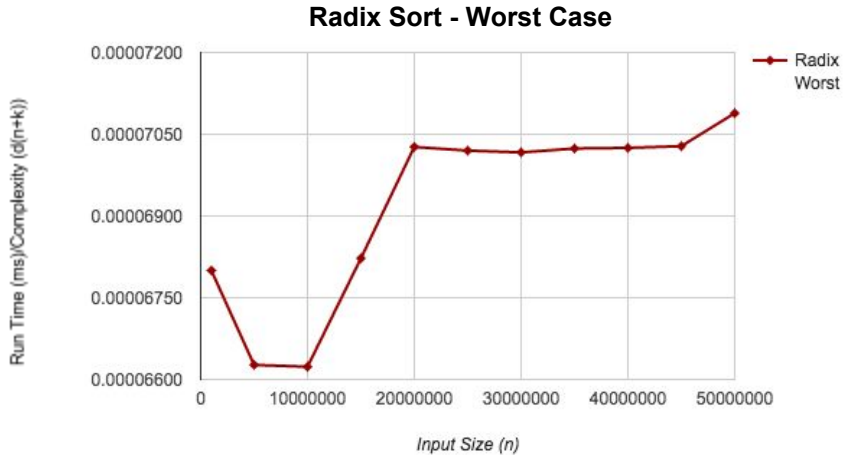


Figure 5M

Figures 5O - 5Q show the analysis of quicksort. The best case and average case showed the most fluctuations, but no steady increases or decreases. Therefore, if the plots were zoomed out with larger y values, the plots would display much straighter lines since the ranges between the y values are small. The plot for worst case was the best for all of the cases. It shows the familiar dip at the beginning due to n being less than n_0, but levels out once n is great enough.

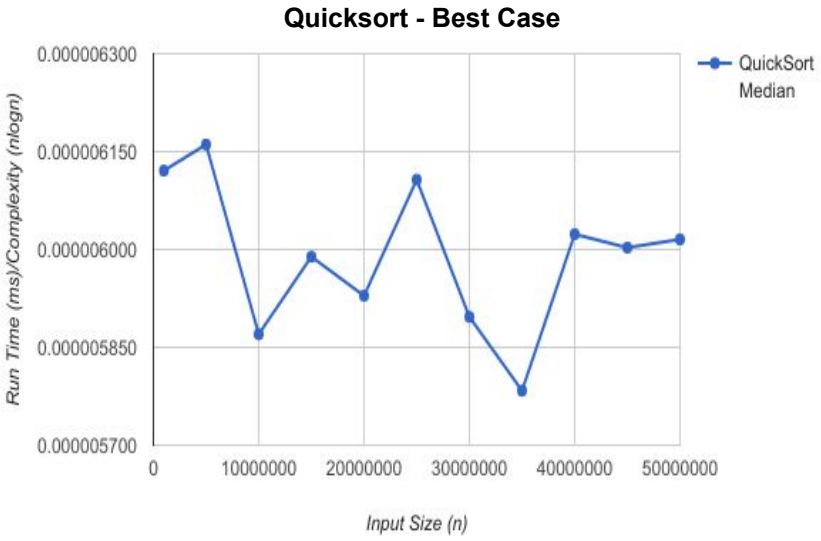


Figure 5O

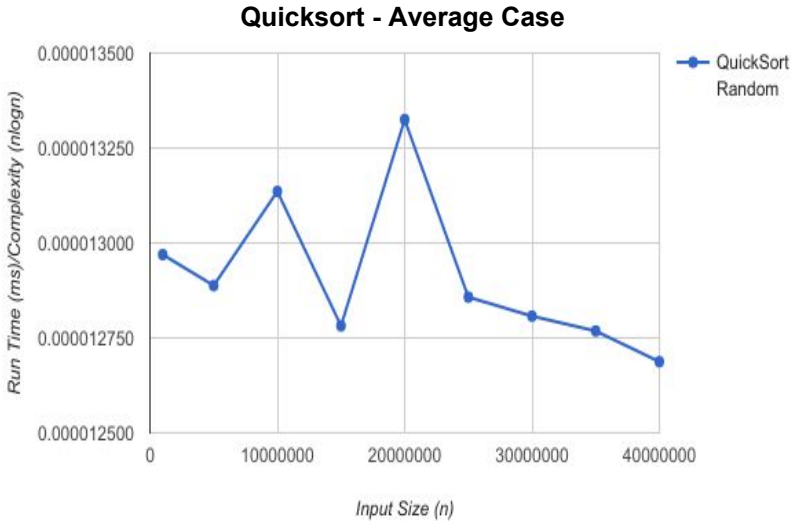


Figure 5P

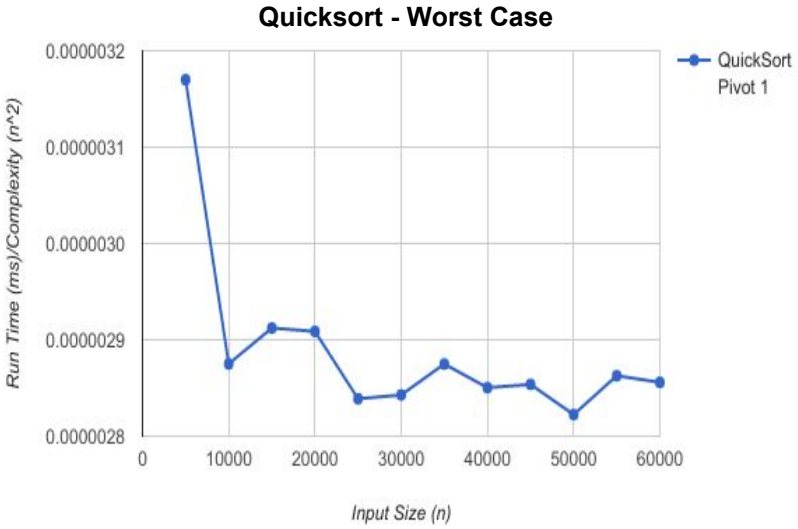


Figure 5Q

Part VI. Comparison Number Analysis

The purpose of this part of the project was to analyze the data in relation to the (Time/Number of Comparisons) vs. Input Size. For the comparison sorts: insertion, merge, quicksort, and heapsort, the results from the graphs indicate that the number of comparisons is a good predictor for determining appropriate execution times. The run times of the sorts compared to the number of comparisons can be said to be directly proportional. That is as the run time increases, the number of comparisons also increases.

From the following plots, most of the sorting procedures gave a relatively straight line with larger n input. All of the sorts were performed with the various inputs of random, increasing, and decreasing input. The input size was chosen based on run time from 5000-15000 ms, trying not to exceed 15000 ms, though there might have been one or two exceptions that were slightly over. All of the plots are zoomed in due to the low value on the y-axis calculated by run time/# comparisons. However, if graphs were zoomed out to show larger values such as integers, the plots would demonstrate straights lines and therefore, show the strong correlation between execution time and the number of comparisons. Most of the graphs show a dip at the beginning due to input size being $n < n_0$. Once the input size becomes large enough, the plots steady out to form a straight line.

The only issue concerning this part of the experiment was running the insertion sort for decreasing input. The first plot used 500,000,000 as the maximum input value, incrementing by 50 million. This resulted in a plot that started low and continually rise. The input size was changed to lower values with a maximum value of 50 million, which resulted in a much better looking and leveled plot, other than the spike in the middle.

Figures 6A - 6C show the results of insertion sort. The plots demonstrate great consistency, with exception of 6C. The spike in the middle of the plot may be due to computer performance.

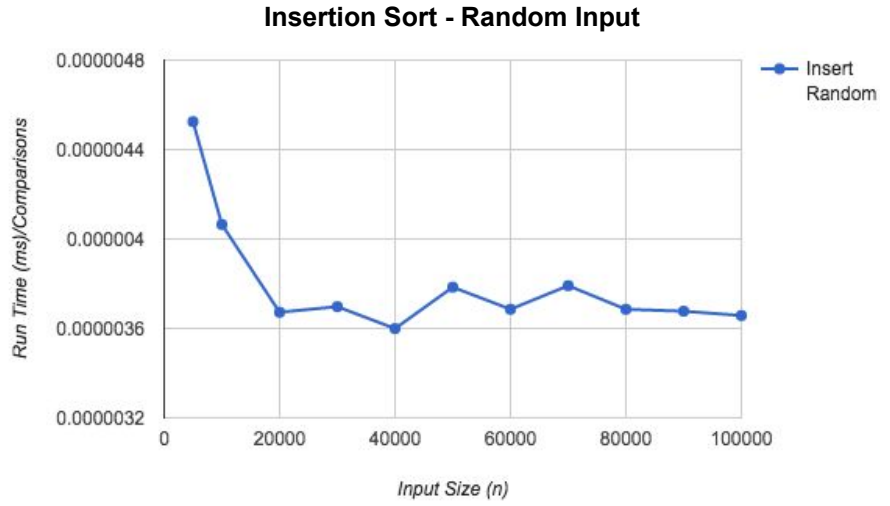


Figure 6A

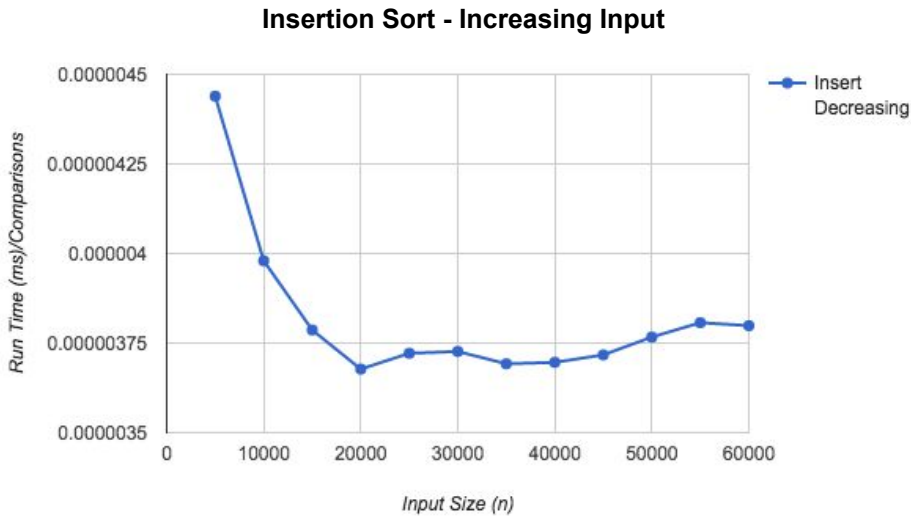


Figure 6B

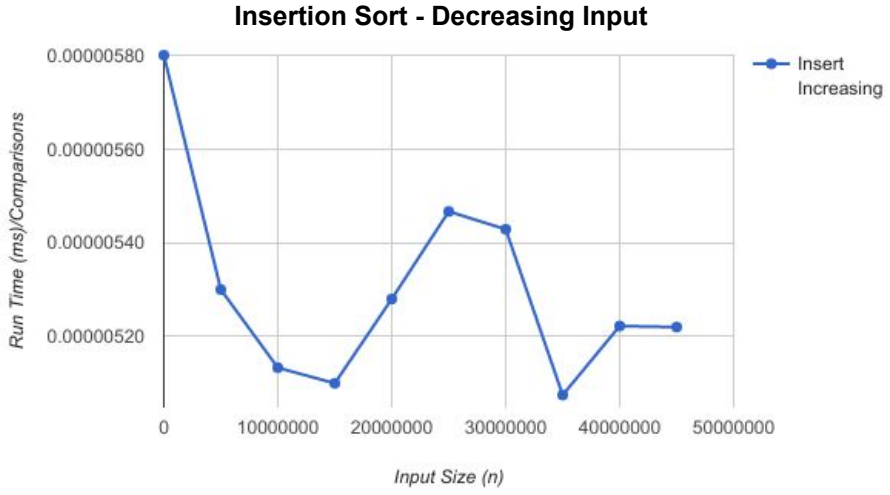


Figure 6C

Figures 6D - 6F show the plots of merge sort

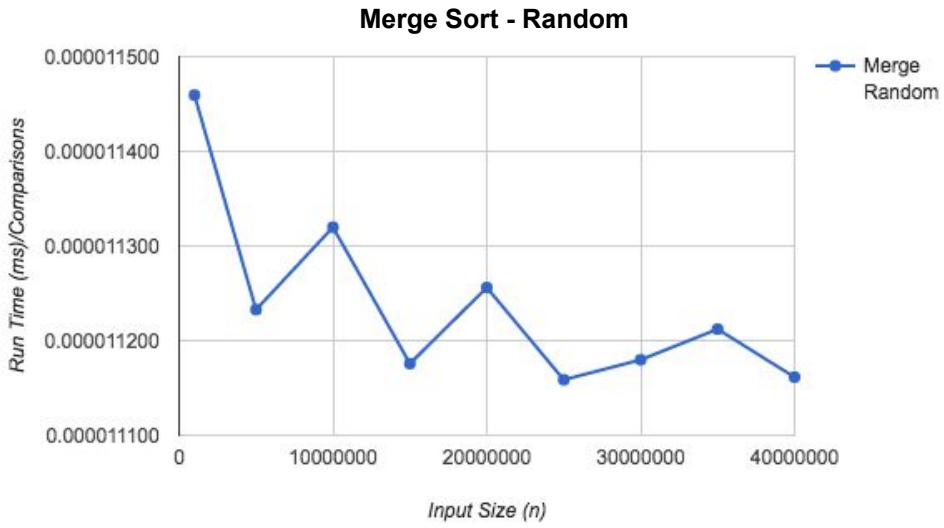


Figure 6D

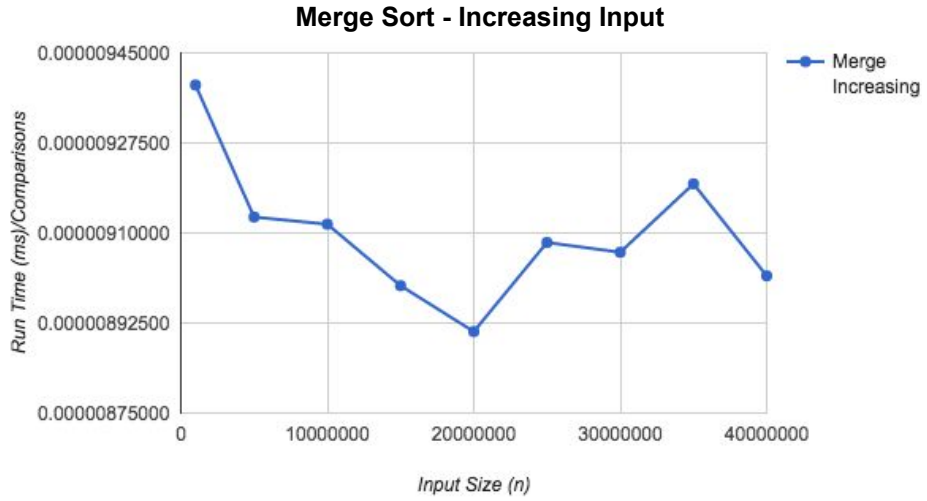


Figure 6E

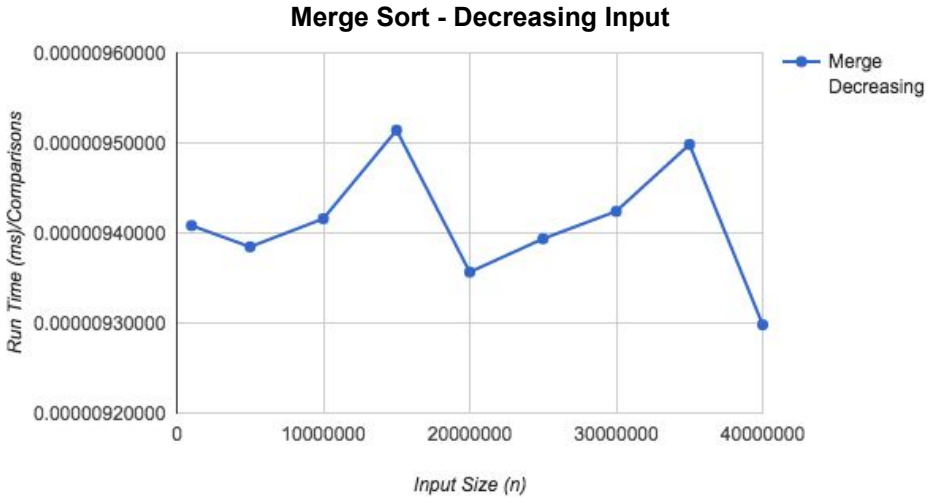


Figure 6F

Figures 6G - 6J show the results of heap sort. Heap sort run with input type of increasing and decreasing order yielded good plots. However, random input shows a rise in the line. Different input size could have been use to try to get a better looking plot, but input size was maintained among all cases. Either way, since heap sort performs O(nlogn) for all cases, the plot for random input should have been more consistent with the other two cases.

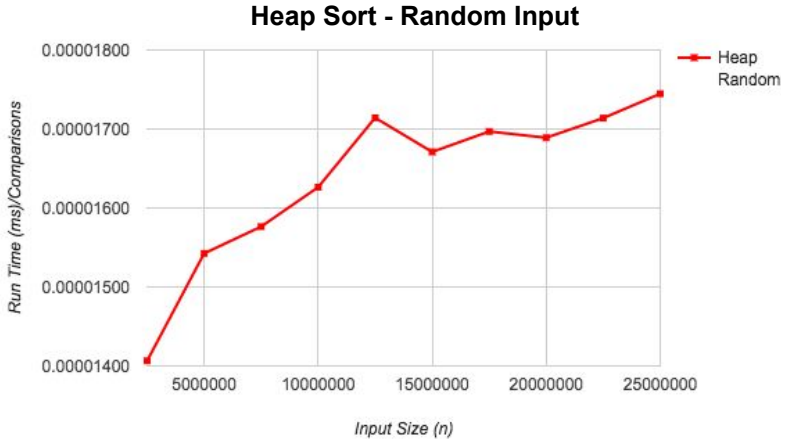


Figure 6G

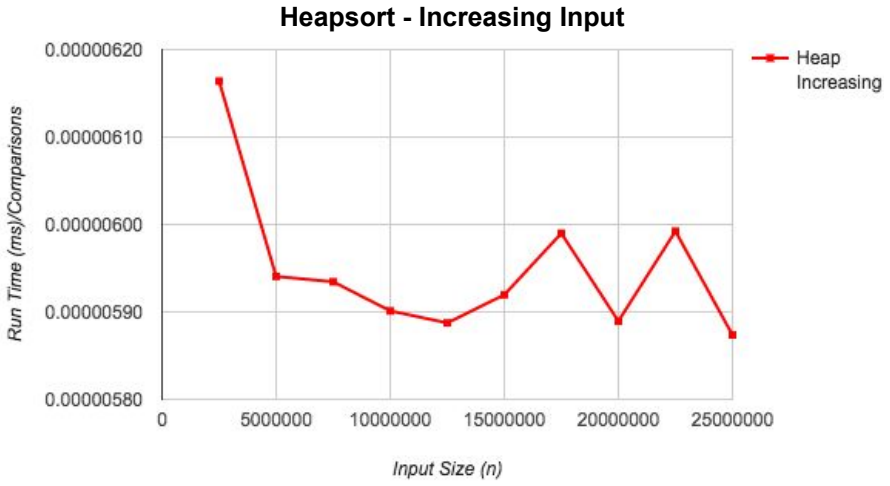


Figure 6H

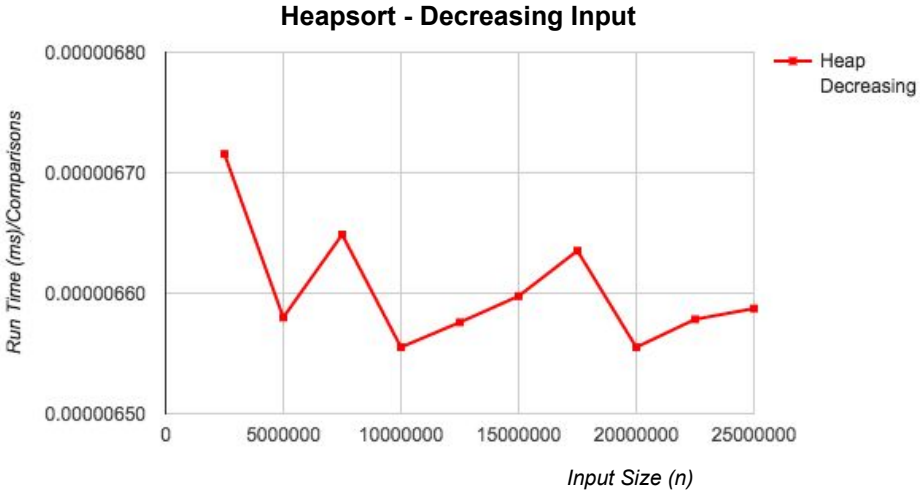


Figure 6J

Figures 6K - 6M show the results of quicksort with the first element as the pivot. Similar to the previous heap sort, the random input resulted in an inconsistent plot. However, the y value range is from 0.000009 - 0.00001, making it very small. Larger values would make the changes hardly noticeable.

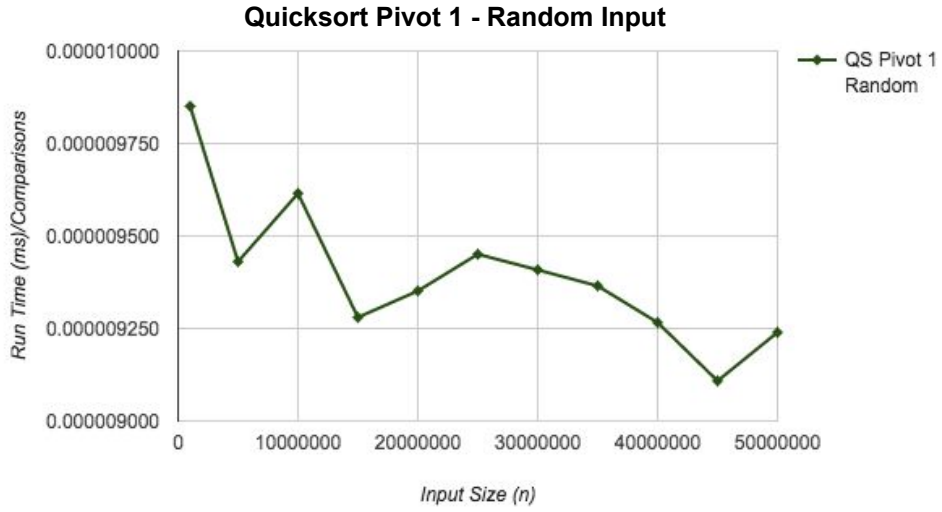


Figure 6K

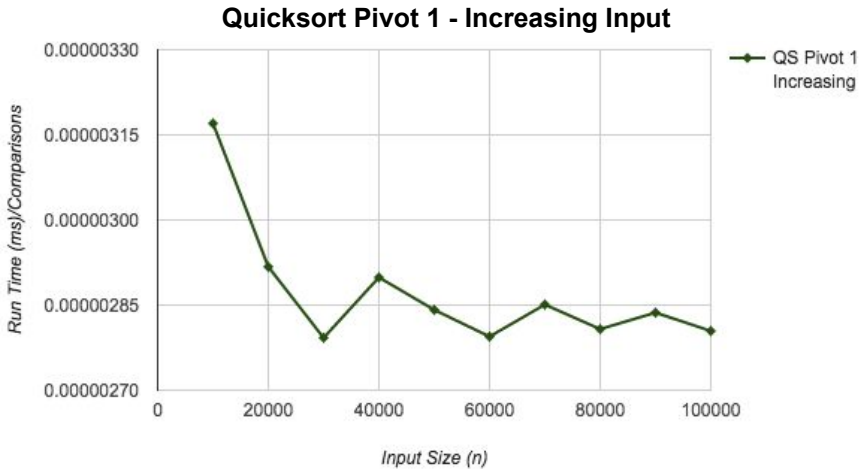


Figure 6L

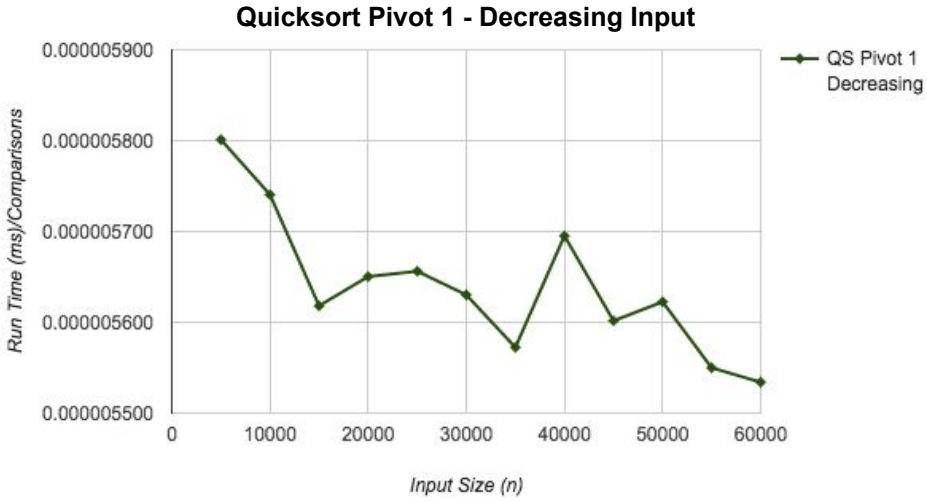


Figure 6M

Figures 6N - 6S are also quicksort, but for the other pivot selections. Similar to the case of quicksort in Figure 6K, the remaining graphs with strong fluctuations still have small ranges in the y values. Therefore, larger plots zoomed out wouldn't show the strong changes.

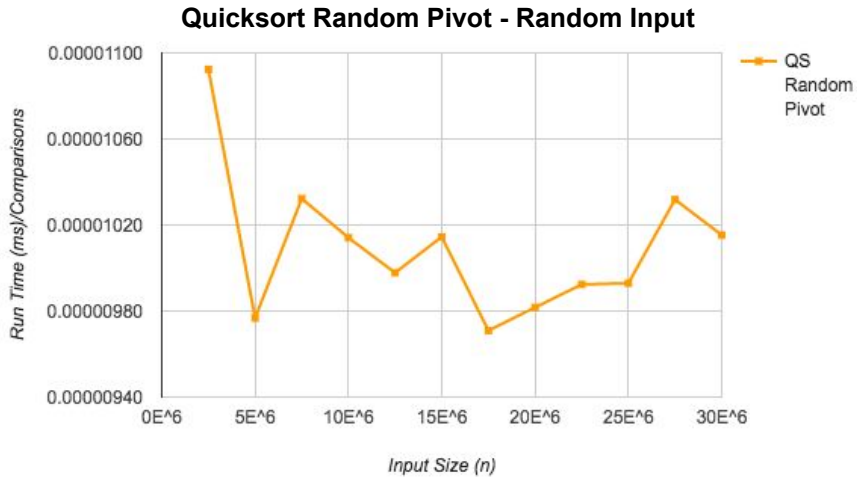


Figure 6N

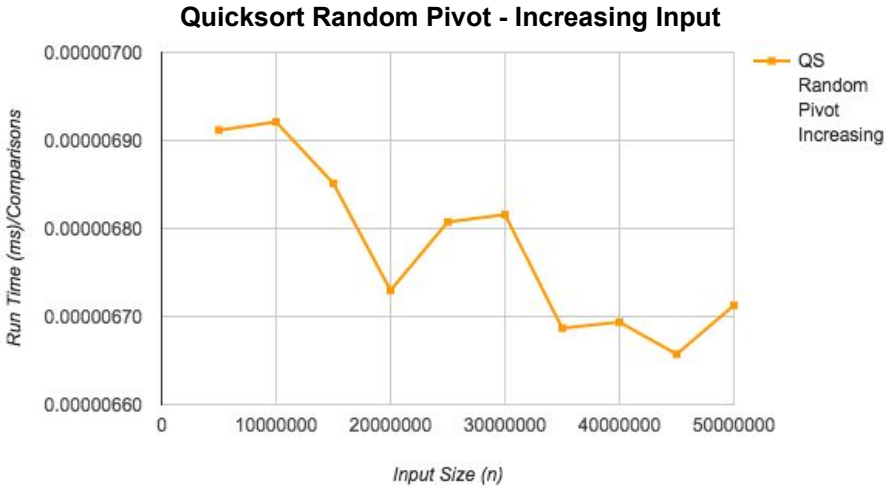


Figure 6O

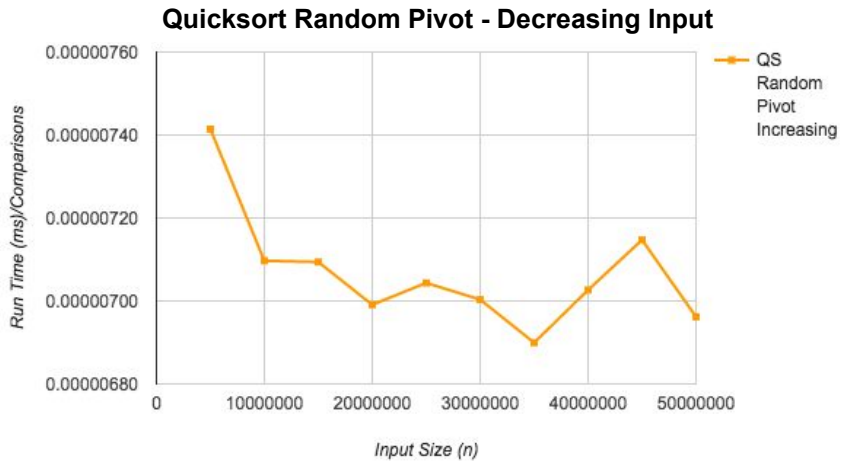


Figure 6P

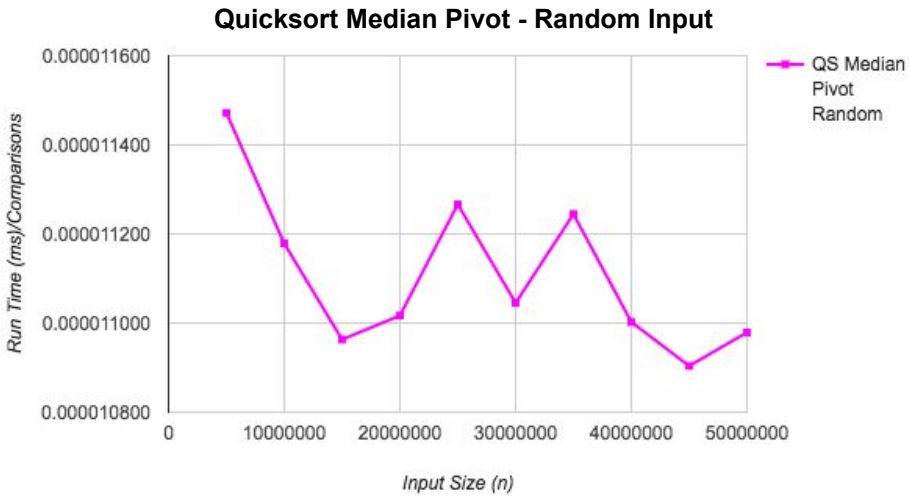


Figure 6Q

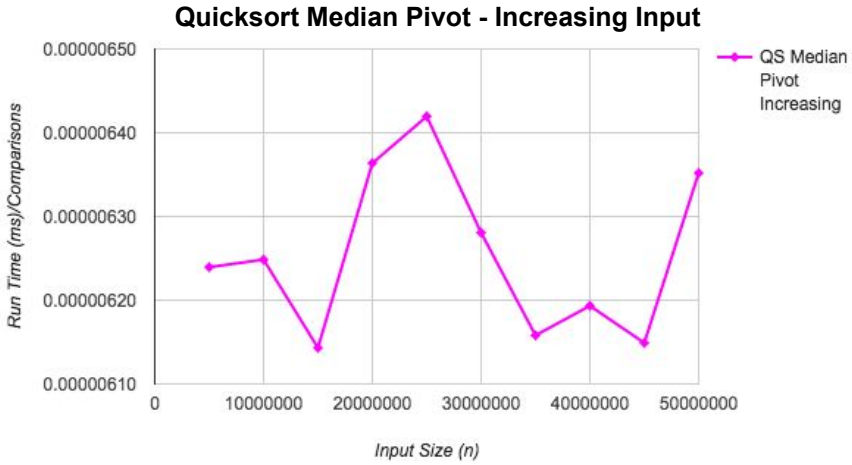


Figure 6R

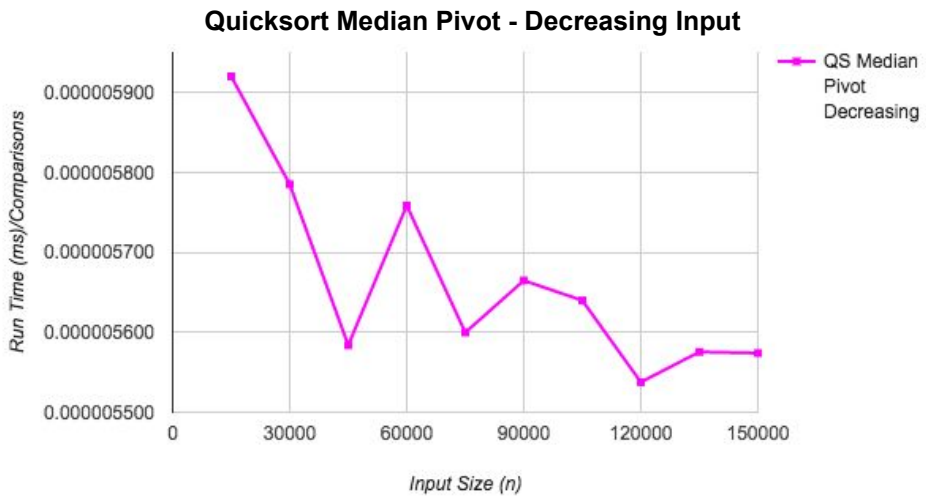

Quicksort Median Pivot - Decreasing Input
Run Time (ms)/Comparisons
0.000005900
0.000005800
0.000005700
0.000005600
0.000005500
0
30000
60000
90000
120000
150000
Input Size (n)
QS Median Pivot Decreasing

Figure 6S

Part VII. Radix Sort Experiment

Objective

Devise an experiment to study the effect of ***b*** and ***r*** on the runtime of Radix sort.

Background

Radix sort functions as a non-comparative sorting algorithm unlike the previous sorts in this project. It uses the counting sort algorithm in order to maintain stability of the elements. This means that the order of the elements are maintained when sorting. Previously stated in Part I, the time complexity of radix sort for best, average, and worst case is $\Theta(d(n + k))$, n = input size, k = to the range of digits, and d = number of digits, or d passes. The purpose of counting sort and radix sort vs. the comparison sorts is to run in linear time. In order for this to occur in radix sort, k must equal $\Theta(n)$, so the time complexity = $\Theta(d*n)$, with d value absorbing into the big-O notation.

Radix Sort works best and maintains stability by working with the least significant digit in the first position, then the next significant digit, until the most significant digit is reached and sorted.

*It should also be noted that radix sort works with an array of strings in digits, making d the the number of characters or digits per string.

Method

Radix sort depends on three parameters, n (size of array), k (size of numbers or character set), and d (the length of the string or number). For the purpose and study of this experiment, the number will be considered a "word" in certain *b*-bits long rather than using decimal 0-9 digits. Each word will be broken into groups of *r*-bit digits. For instance, a b = 32-bit word will be broken into r = 8-bit digits, resulting in d = b/r = 32/8 = 4 passes. The number of passes is a feature in radix sort and tells how many times the count will run in total. Words or numbers are divided in order to determine the appropriate number of passes.

Figure 7A shows a 32-bit word divided by 8 bit digits = 32/8 = 4 passes.

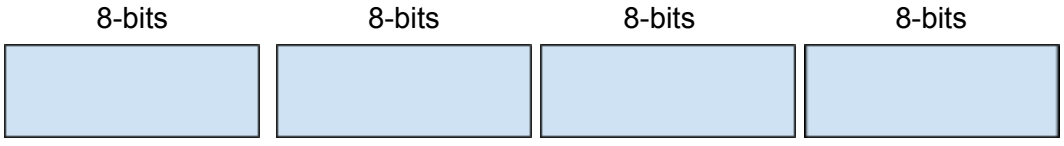
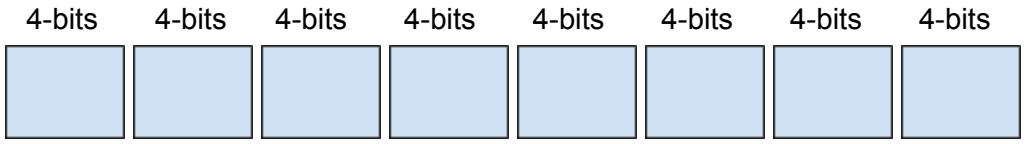


Figure 7A

Since the value of r = 8, and the equation $2^r - 1$ gives us the range k of the inputs, then k = $2^r - 1 = 2^8 - 1$ = 255 digit range. More examples can be determined with various b-bit words, divided by various r-bit digits. Therefore, giving different d = b/r values, or number of passes.

Figure 7B provides the test with the same word size of 32-bits, but divided by the r value = 4. This results in d = b/r = 32/4 = 8 passes with each segment being 4-bits long.

Figure 7B



To study the effects of b and r on run time, analysis is to be conducted with various r-bits on a various input size n. Each of the inputs will contain words that are b-bits long. The input size will begin low and increment steadily by 1,000 until the sort loses its linear complexity. Also, the b-bit word will have to be large enough to be able to be divided by the multiple r-bits. The example in Figure 7A is a 32-bit word, and can only be divided by 32, 16, 8, 4, 2, and 1. Therefore, the possible d values would be 1, 2, 4, 8, 16, and 32 passes.

It is important to understand the effects that the b/r = d value has on the k value, and the entire problem. Essentially, the larger the d value, the smaller the k value. So the tradeoff needs to be analyzed, for instance, if the 32-bit word was divided into 8-bits = 4 passes, the k value would be 255 digits. In contrast, if the 32-bit word was divided into 16-bits = 2 passes, the k value would be 2^{16} digits, a much larger k value. But overall, the n is a big factor as to what values b and r can be.

The main experiment would be to create an algorithm that takes the various input sizes of a various large bit values. So, for the case of an example, the input values are 64-bit words. The algorithm will therefore conduct the following process with r values of 1, 2, 4, 8, 16, 32, 64:

1) b/r = 64/1 = 64 passes
2) b/r = 64/2 = 32 passes
3) b/r = 64/4 = 16 passes
4) b/r = 64/8 = 8 passes
5) b/r = 64/16 = 4 passes
6) b/r = 64/32 = 2 passes
7) b/r = 64/64 = 1 passes

The run times for the experiment will show how the selected values affect each other and compare the run times side-by-side for all n values. The program will allow the user to change the input size, word size, and r-value. When the sort begins to lose its linear complexity, the program will terminate.

Analysis

The analysis of radix sort should demonstrate its time complexity $T(n,b) = \Theta(b/r(n + 2^r))$. Therefore, r is the key component to minimizing T(n,b). By increasing the r value, the sort will run with fewer passes. However, to maintain the linear complexity, it is important to avoid making r>>lgn. If this were the case, then that would make 2^r much larger than n. The algorithm would be setup to accept all possible values of n, b, and r. The run times will show that b and r can be as large as possible, but within the bounds of the

input size. The results of the run time will show that there is a stronger correlation between the input size n and the value of r, rather than the value of r and the value of b.

Finding $r \approx \lg n$ => b/r = b/lgn which will perform at $\Theta(d*n) = \Theta(b/r * n) = (b/\lg n * n)$

Program Examples:

1) Sorting 2^8 = 256 numbers, 32-bits:
    - *n* = 2^8 numbers
    - r = lg n = lg 2^8 = 8 bit digits
    - k = 2^r-1 = 2^8-1 ≈ 256 range
    - b = 32-bit words
    - d = b/r = 32/8 = 4 digits
    - 4 digits requires 2 passes to sort
    - Thus, Radix Sort performance: 4*n*

2) 1,000,000 numbers, each 64 bits :
    - 2^20 numbers, each of 64 bits
    - n = 2^20 numbers
    - r = lg n = lg 2^20 = 20-bit digits
    - k = 2^r-1 = 2^20-1 ≈ 1,000,000 range
    - b = 64-bit words
    - d = b/r = 64/20 = 4 digits
    - 4 digits requires 4 passes
    - Radix Sort performance: 4*n*