CSCE 221 Cover Page

Programming Assignment #4

Due Date: Friday November 15, 11:59pm

Submit this cover page along with your report

First Name: Bryant Last Name: Passage UIN: 326009948

Any assignment turned in without a fully completed coverpage will receive ZERO POINTS.

Please list all below all sources (people, books, webpages, etc) consulted regarding this assignment:

CSCE 221 Students	Other People	Printed Material	Web Material (URL)	Other
1. Josh	1.	1.	1. stackoverflow.com	1.
2. Elaine	2.	2.	2. geekforgeek.com	2.
3.	3.	3.	3.	3.
4.	4.	4.	4.	4.
5.	5.	5.	5.	5.

Recall that University Regulations, Section 42, define scholastic dishonesty to include acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. Disciplinary actions range from grade penalties to expulsion. Please consult the Aggie Honor System Office for additional information regarding academic misconduct – it is your responsibility to understand what constitutes academic misconduct and to ensure that you do not commit it.

I certify that I have listed above all the sources that I consulted regarding this assignment, and that I have not received nor given any assistance that is contrary to the letter or the spirit of the collaboration guidelines for this assignment.

Today's Date: 11/17/19

Printed Name (in lieu of a signature): Bryant Passage

Sorting Algorithms and Time Complexities

Bryant Passage

Introduction:

In this project, several types of sorting algorithms were tested and timed in order to find the most efficient sorting algorithm based on three different input lists: a totally random list, a sorted list, and a reverse sorted list. The types of sorting algorithms tested during this project include bubble sort, merge sort, heap sort, quick sort, and, as an extra, radix sort was also tested.

Theoretical Analysis:

Inside bubble sort's implementation, there are two loops that are required in order to completely sort a list using bubble sort. Bubble sort works by swapping adjacent elements according to whichever element is greater than the other. For this to work, bubble sort takes the first element and compares it to the second element. If the second element is smaller than the first element, then those two elements swap so that the greater element is closer to the end of the list and becomes the second element. Bubble sort then takes the new second element and compares it with the third element. This process continues until the nested loop reaches the end of the list. Then, the process reverts back to the first loop and the first element of the list is checked again with the second element. This process is repeated until the whole list is sorted. Measuring the time complexity, the outer loop will always run n times because of the fact that each element needs to be compared at least once. The inner loop will run $n - i$ times where i is the number of times the outer loop runs. This is because bubble sort always takes the greatest element in the list and swaps it until it is at the back of the array. Since the largest element is located at the back of the array after every iteration of the outer loop, there is no need to loop through the entire array again. Therefore, the number of times the second loop iterates decreases by one every time the outer loop iterates. Taking this into account, bubble sort requires n + (n-1) + (n-2) + (n-3) + … + (n-n) time to sort a list. Upper-bounding this time complexity will achieve a time complexity of $O(n^2)$. Since bubble sort will always compare each element in a list and is also not a recursive process that relies on previous executions of itself, bubble sort will always run based on the number of elements, n, in a list. Therefore, bubble sort will always run in $O(n^2)$ time in all three input cases which are the random list, sorted list, and reverse sorted list.

Merge sort is a divide and conquer algorithm that relies on creating subproblems in order to more efficiently reach the end goal. Merge sort essentially splits the list into two subproblems after each call and will recursively continue to split each subproblem into two smaller problems until only one element is remaining in the list (the other elements in the list do not magically disappear; the expression is in relation to the scope of each recursive call of merge sort). When there is only one element remaining in the subproblem, the merge aspect of merge sort executes. This essentially takes two subproblems and combines them into one list again. However, while combining, the first elements of each subproblem/sub-list are compared. The smaller element of the two will be popped from the respective sub-list and then appended to the new list. Then the first elements of each sub-list are compared again. This process continues until either both subproblems are empty or one subproblem is empty. If the latter occurs, then the subproblem with leftover elements will be appended in the order that they are already in (they will already be sorted due to previous merge functions that occurred before). Once both subproblems are empty, the function finishes and returns to the next function in stack to be executed which is another merge function on two other subproblems. This process will continue until all subproblems/sub-lists have been reached and combined. The result of this process is a sorted list that is also stable. The time complexity of merge sort is more difficult to understand due to its divide and conquer method of solving a sorting problem. When looking at just the division part of merge sort, merge sort recursively divides a list into two subproblems/sub-lists until each subproblem only contains one element. Therefore, if a list contains n elements, then the number of executions can be

calculated as $n = 2^k$ where k is the number of times the list is divided into two sub-lists. This means that $k = log_2(n)$. Next, once every subproblem has one element, merging two sub-lists requires $O(n)$ time. This is because the first two elements of each subproblem needs to be compared and the process continues until one of the two sub-lists have no remaining elements. The number of merges that occur is also the number of executions of dividing the list into two subproblems which was found earlier. Therefore $O(n) * log_2(n)$ yields a time complexity of $O(nlogn)$ for the merging aspect of merge sort. Combining the process of dividing and merging, the time complexity yields to be $O(logn + nlogn)$ which simplifies to just $O(nlogn)$. Therefore, the time complexity of merge sort is $O(nlogn)$. Merge sort also only cares about the number of elements in the list to be sorted rather than the elements of the list. Since merge sort will always divide the problem into two equal subproblems (or two subproblems with one of the subproblems containing one more element than the other), merge sort's time complexity only relies on the number of elements in the list. Therefore, merge sort should run equally on the random list, the already sorted list, and the reverse sorted list which is $O(nlogn)$.

Heap sort utilizes the heap data structure and the priority queue implementation that were discovered in the previous project. From the previous project, it was discovered that a min-heap takes $O(logn)$ time to insert and element and $O(logn)$ to remove the minimum element. Inserting an element requires $O(logn)$ time because the heap must rearrange itself by performing up-heaps in order to maintain the ordering of the binary tree and avoid any violations of the parent-child relationship where the parent must be smaller than its two children. The same can be understood when removing the minimum element as well. Since the minimum element is located at the root of the binary heap tree, the last element of the tree replaces the root. However, most of the time this also violates the min-heap tree and, therefore, the heap must perform down-heaps in order to maintain the parent-child relationship. Once the heap is created, heap sort will remove all elements in the heap by removing the minimum after every iteration until all elements in the heap are removed. The minimum removed elements are stored in a new array in the order of the removal which means that the new list is sorted from smallest element to largest. Since heap sort relies on the inserting of elements in a heap and then removing the root until there are no more vertices in the binary tree, the time complexity for n elements for heap sort is $O(nlogn + nlogn)$ as this accounts for n insertions and n removals. This further simplifies to $O(nlogn)$ as the total time complexity to sort a list using heap sort. The heap sort used in this project was a bottom up heap implementation which requires constant rearranging of the heap whenever an element is inserted or removed from the heap. This means that there should be no difference when performing heap sort on a random list, a sorted list, or a reverse sorted list.

Quick sort is another divide and conquer algorithm, but unlike merge sort, quick sort divides the problem into three sub-problems: a less than list, an equal to list, and a greater than list. Merge sort first takes an arbitrary point (called the pivot) to partition the list. For this project's implementation of quick sort, the pivot is always the last element of the list. Once the pivot is chosen, a linear comparison will execute through the list and find elements that are greater than or equal to, or less than the pivot's value. The elements that are greater than or equal to the pivot's value are inserted into one array which will be called the "greater than array" and the elements that are less than the pivot's value are inserted into another array which can be called the "less than array." After the three partitions have been established, the quick sort function will be executed again, however, limited only to one of the partitions (in this case, the less than array is always visited first before the greater than array). In quick sort, the "divide" part is the partitioning of the array into smaller problems, however, unlike merge sort, these partitions are not always of equal length and rather can be very different in size. The "conquering" part is the appending of each subproblem/sub-list after every subproblem has been partitioned until there is only one or no elements in each partition. This partitioning algorithm will naturally sort the each subproblem as it rearranges the elements of the subproblem based on its relation to the pivot of each

subproblem. Then, once there is either only one or no elements in each subproblem, each partition of the subproblem (the less than, greater than, and equal to sub-lists) are appended to each other by order of the less than list first, then the equal to list, and lastly the greater than list. This is similar to the "merging" function of merge sort. Once the all subproblems have been appended together, the list should be in sorted order. The time complexity of quick sort depends on how well the pivot is chosen during each call of quick sort. If very unlucky, the pivot is chosen to be the extreme values of a list (either the greatest element or the smallest element). This results in the execution of each subproblem by *n-i* times where *i* is the number of partitions performed. Enumerating this process, if there are *n* elements in a list and the pivot chosen for each time the list is partitioned is either the largest or smallest element in the list, then quick sort will perform *n* comparisons for the first partition and then *n-1* comparisons for the second partition and so on. This gives the expression $T(n) = n + (n-1) + (n-2) + \dots (n-n)$. Since this expression will run *n* times, then the time complexity can be upper-bounded as $O(n^2)$. This is the worst-case time complexity of quick sort. If the pivot is well chosen every time the list is partitioned, then the time complexity of quick sort performs similarly to merge sort because there are two subproblems with more than one element in each of them after each partition rather than just one subproblem with just one less element in it. A good pivot position after each partition will result in a time complexity of *O(nlogn)* to sort through *n* elements. Because of the notion that quick sort relies on "lucky" pivot positions, quick sort performs better in a truly randomized list. This means that a sorted list and a reverse sorted list will perform very poorly due to the fact that the pivot that is chosen after every partition is either the largest or smallest element of that subproblem. This will result in the worst-case time complexity of $O(n^2)$ time when sorting through an already sorted list or a reverse sorted list. In a truly randomized list, quick sort will perform in *O(nlogn)* time.

Radix sort in this project is an extra sort that was tested in conjunction with the other sorts to find the best sorting algorithm for these test sets. Please note that radix sort is NOT a comparison sort and is much harder to implement when the test cases are anything other than a list of integers or characters. Since radix sort is not a comparison sort, it is able to break the lower bound of *O(nlogn)* time complexity that comparison sorts are limited to. Please read the report also written by me solely on radix sort and its performance. Long story short, radix sort is able to run in *O(n)* time with the right test conditions.

Experimental Setup:

A. Machine Specifications:
   - Custom built PC
     - Ryzen 7 3700X @ 3.6GHz Base → 4.4Ghz Boost[1]
     - ASUS X470-F Gaming Motherboard
     - 16 GB Trident Z RGB 3200MHz DDR4 RAM
     - Nvidia GeForce GTX 1070 Founders Edition

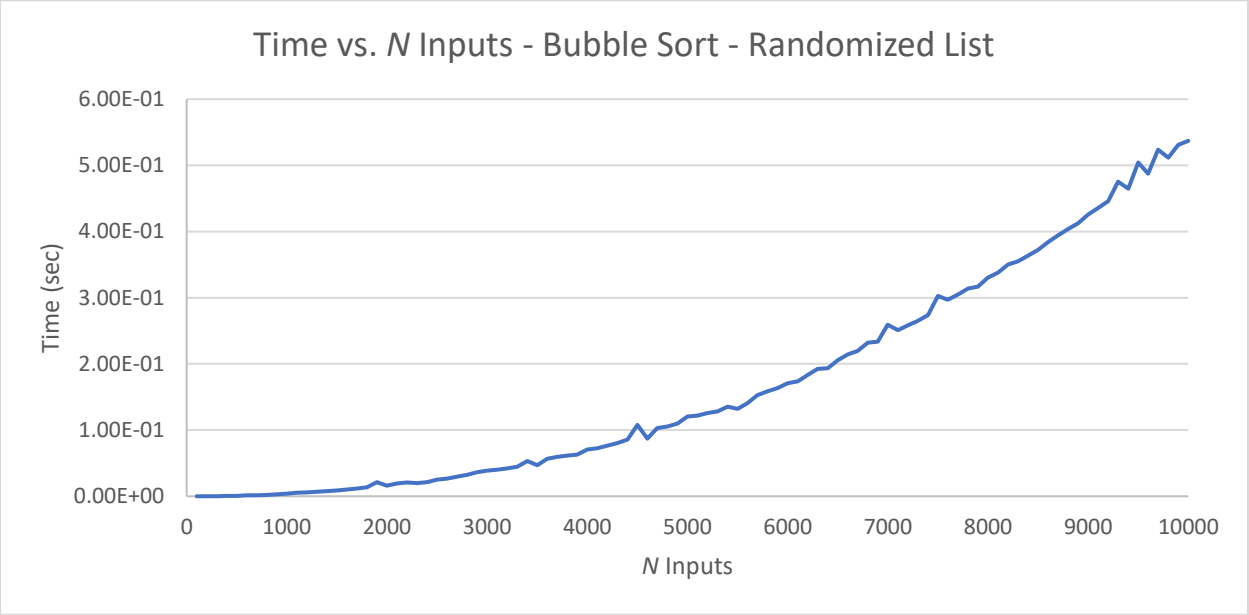
B. Test Setup:
   - Each sort was tested with three test cases:
     - Randomized list using the rand() function
     - Previously sorted list
     - Reverse sorted list

[1] Note: The boost specifications of this CPU vary throughout the time the program runs. Therefore, the comparison should only be made relative to the other tests conducted with the same test bench.

- Each list starts at 100 elements and increases by 100 elements after every successful sort for 100 iterations
- Total number of elements to sort during last iteration is 10,000 elements
- Every sort got its own list; for the randomized list, a random number was stored and inserted into every sort's list so that every list was identical
- All sorts utilized standard C++ arrays except for quick sort which utilized vectors for its subroutine (however, it still utilized standard arrays for test inputs)

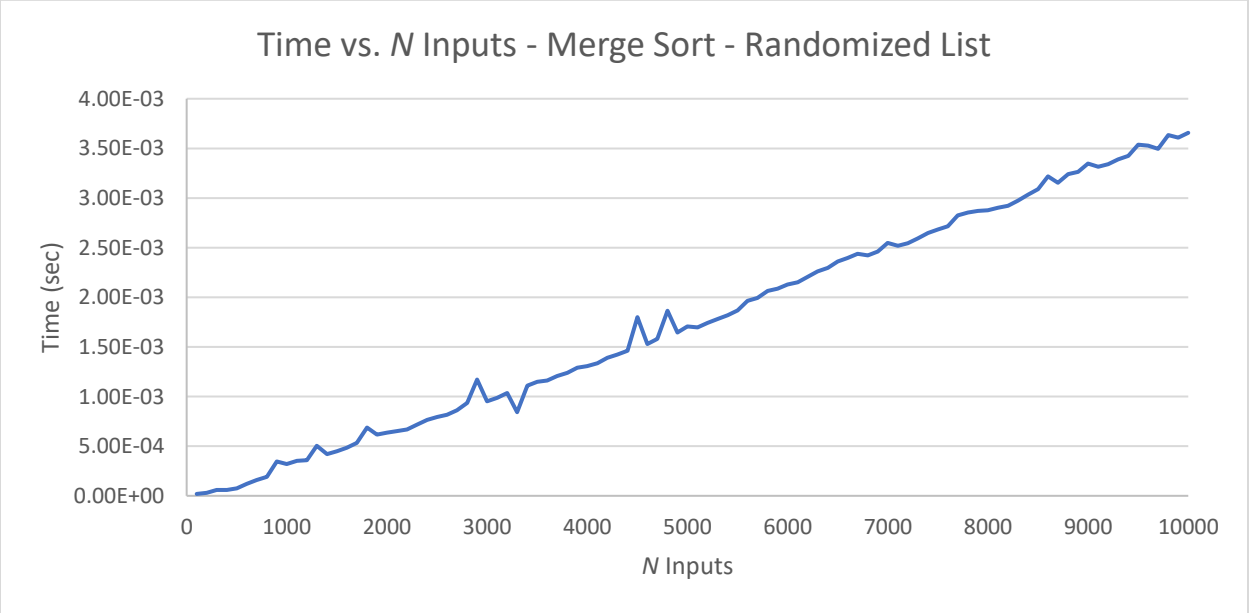
Experimental Results:

Bubble Sort – Randomized List:



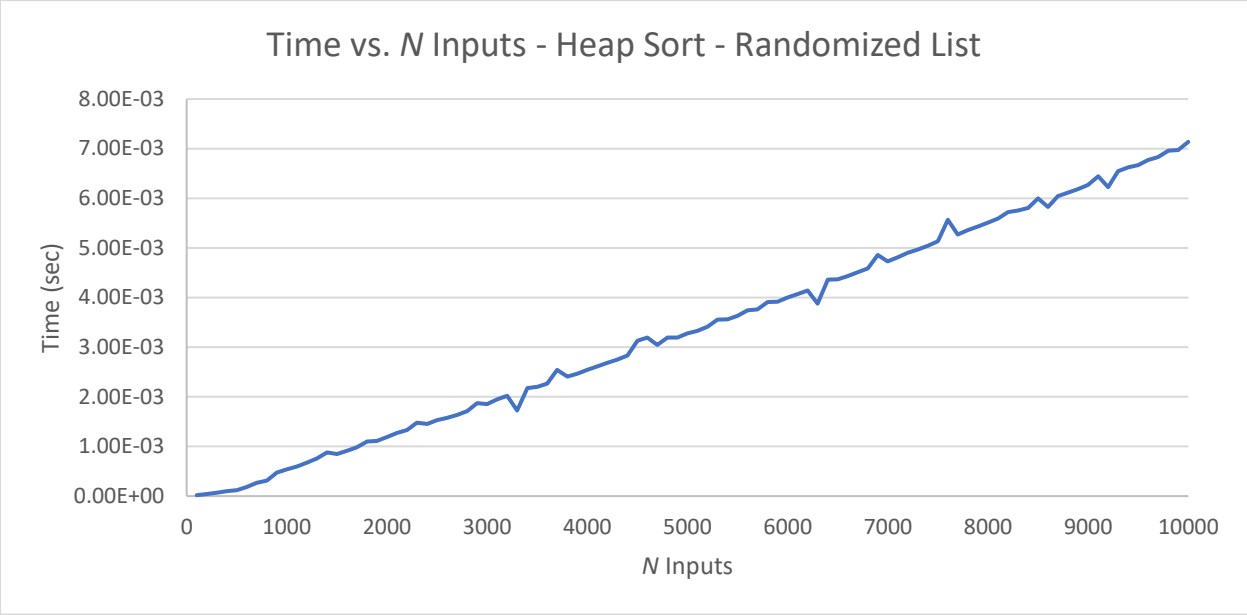
Here, bubble sort can be seen with an upwards curve indicating that bubble sort does not perform in linear time as the number of inputs increased. Note that this is for a randomized list.

Merge Sort – Randomized List:



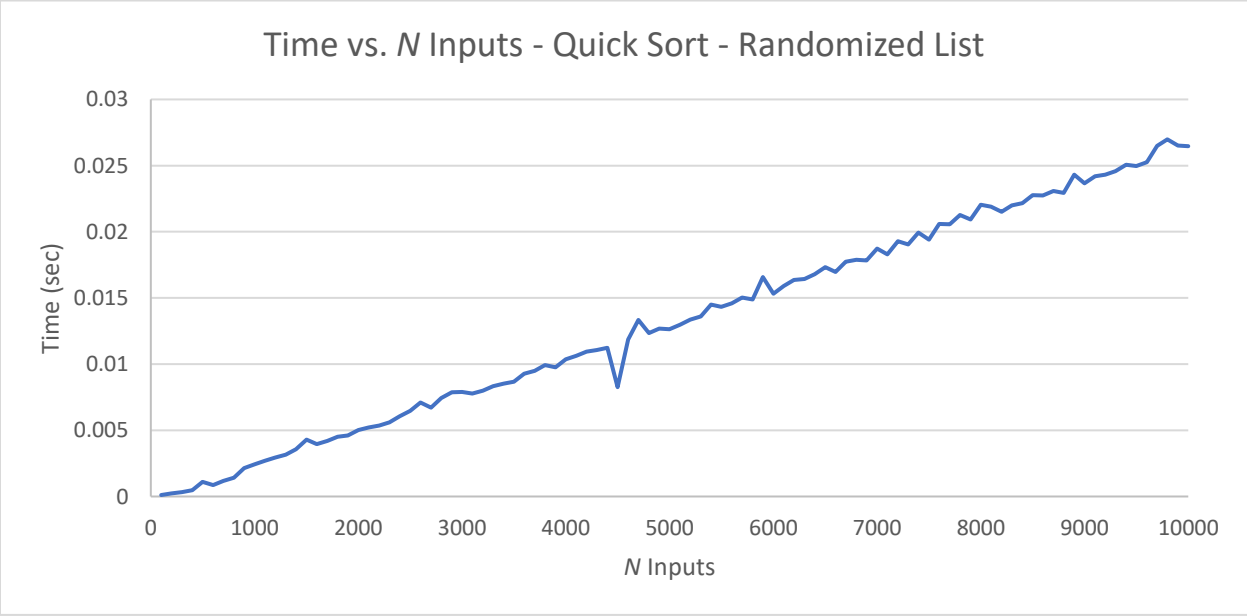
Merge sort appears to run in linear time due to the linear nature of the plot above, however, merge sort is running in linearithmic (nlogn) time. Merge sort here is sorting a randomized list of increasing elements.

Heap Sort – Randomized List:



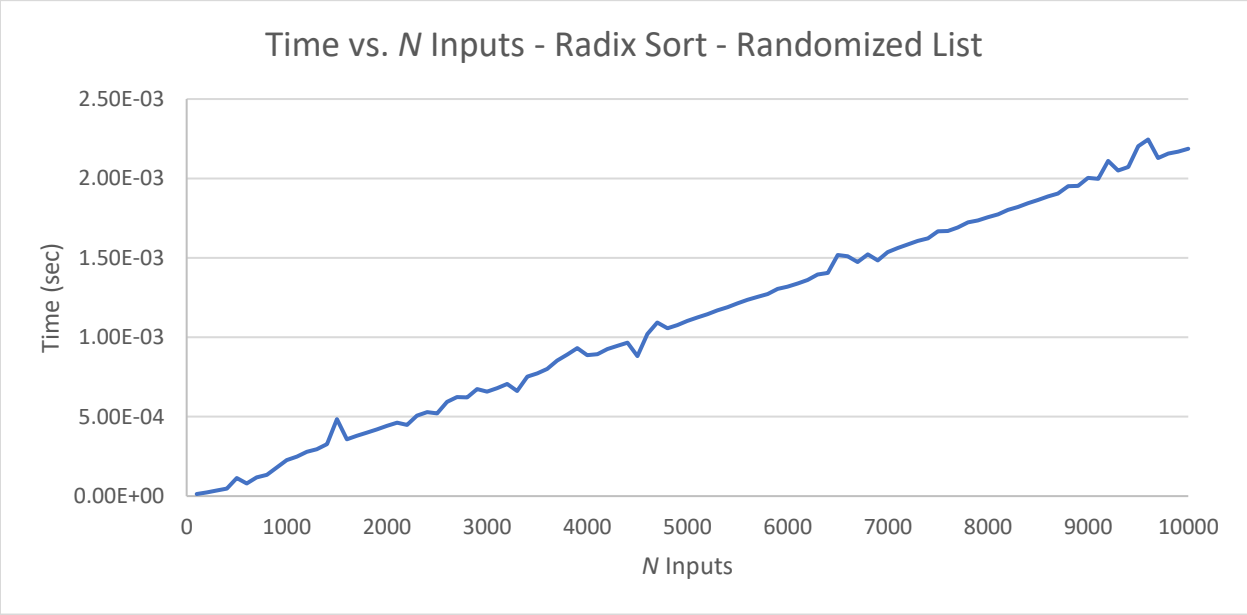
Heap sort's graph looks very similar to merge sort's graph, however, there is a hidden constant multiple in time complexity analysis that is in favor of merge sort rather than heap sort. Here, to sort the same number of elements takes longer using heap sort than merge sort.

Quick Sort – Randomized List:



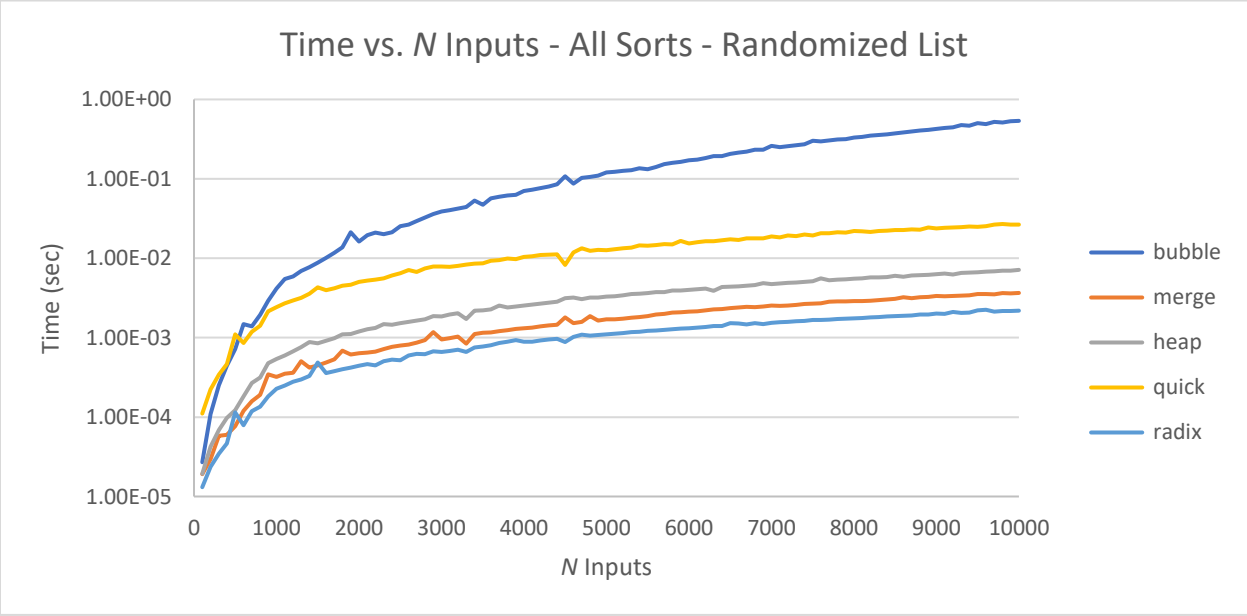
Quick sort's graph looks similar to the previous two sorting graphs, however, quick sort here takes even longer than heap sort. This could be due to the smaller randomized list or possibly multiple common elements present in the list.

Radix Sort – Randomized List:



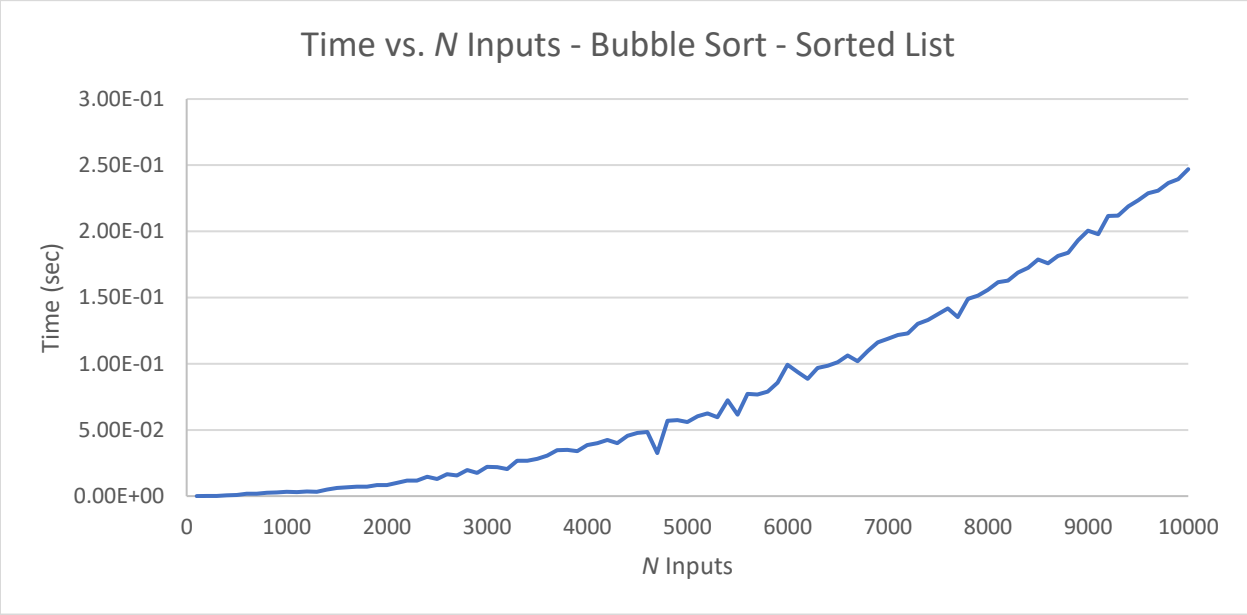
Radix sort appears to run the fastest compared to the other sorts tested in this project, sorting 10,000 elements in less than 2.5 milliseconds. Here, radix sort appears to be running either linearly or in linearithmic time.

All Sorts – Randomized List:



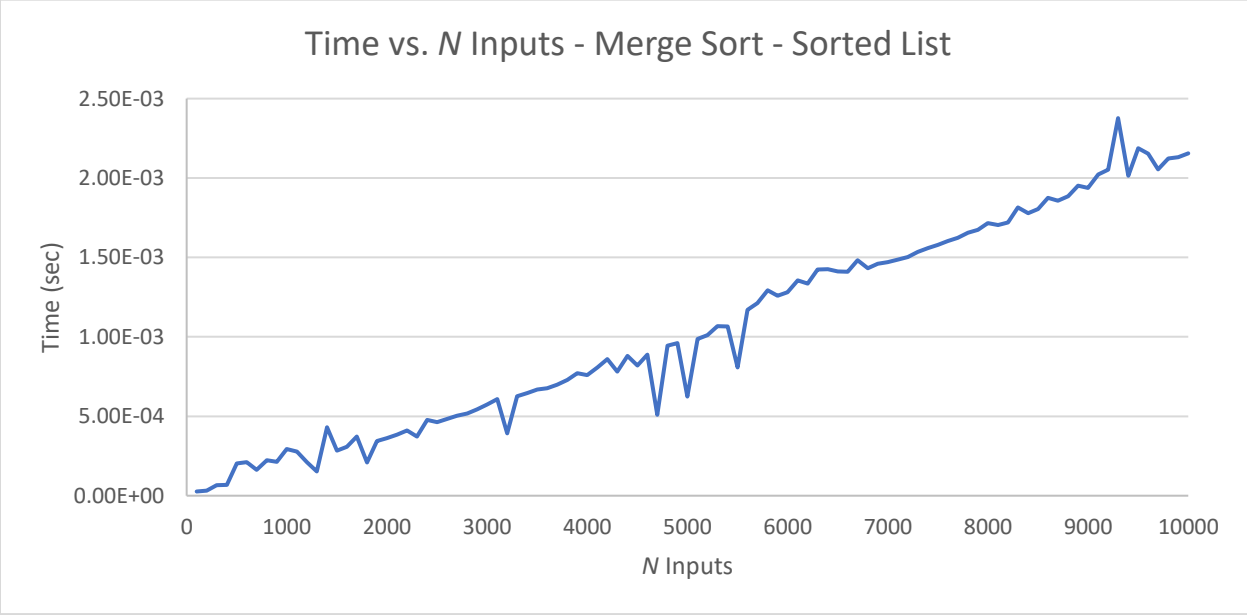
In this graph, all sorts are depicted to show their running time versus each other. Note that the y-axis scale is logarithmic. Here it can be seen that radix sort performs the fastest followed by merge, heap, quick, and bubble sort.

Bubble Sort – Previously Sorted List:



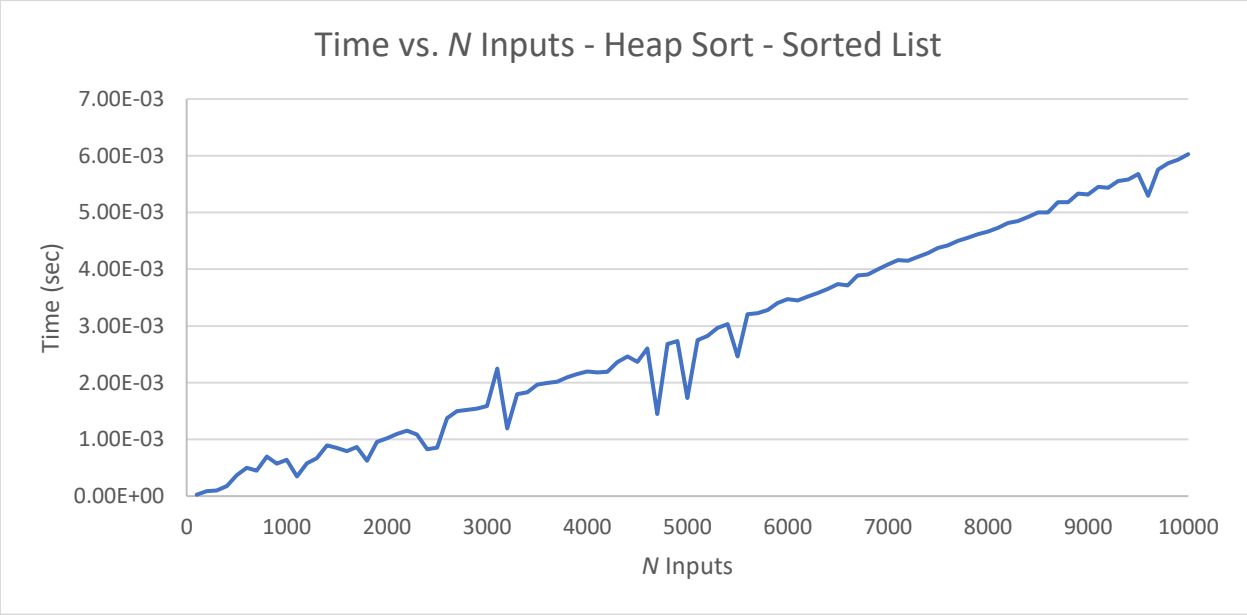
Bubble sort still appears to run in $O(n^2)$ time, however, with an already sorted list, bubble sort runs faster than when it was given a randomized list.

Merge Sort – Previously Sorted List:



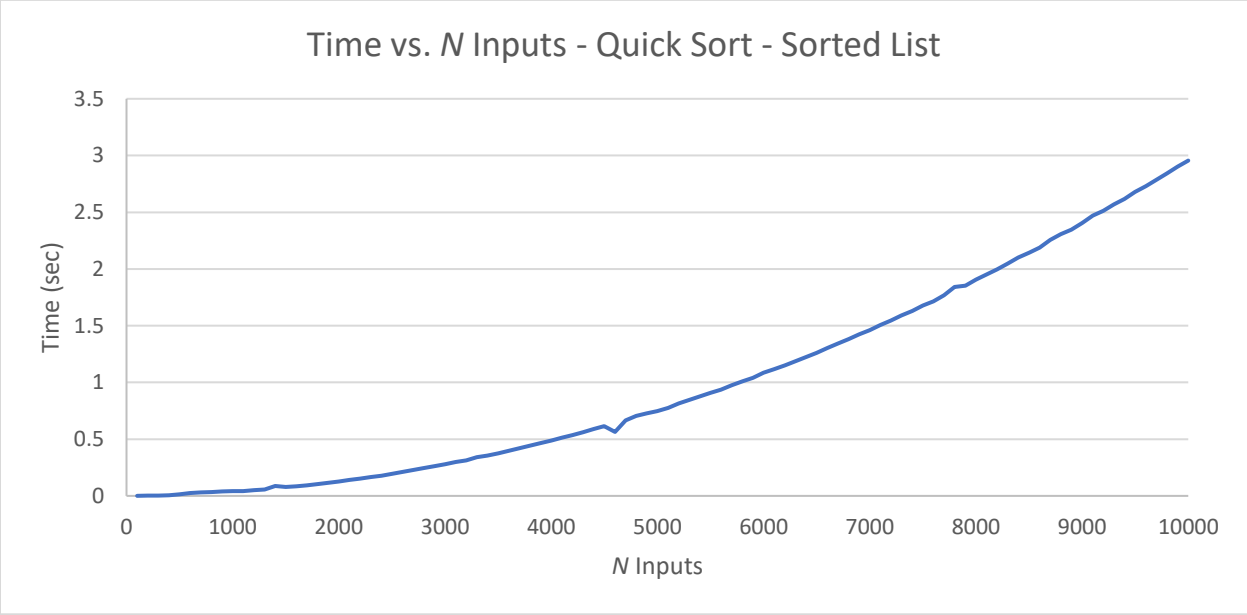
Heap sort also appears to have the same time complexity with a sorted list and a randomized list. We also see that merge sort runs slightly faster with a sorted list than with a randomized list.

Heap Sort – Previously Sorted List:



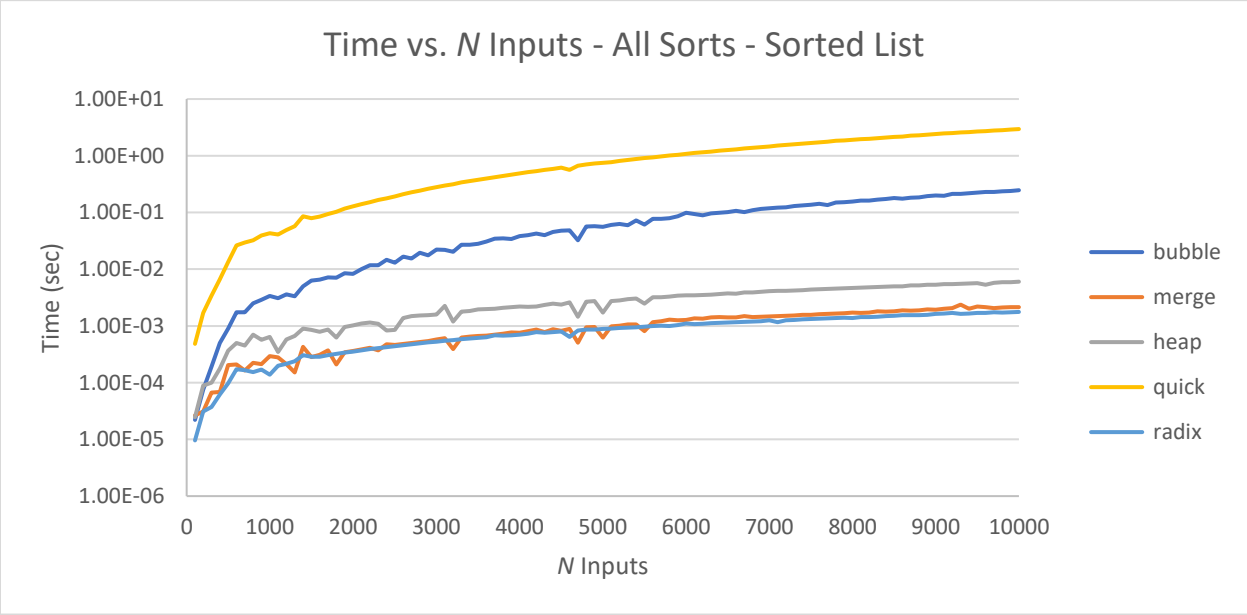
Heap sort appears to run slightly faster with a sorted list than the randomized list.

Quick Sort – Previously Sorted List:



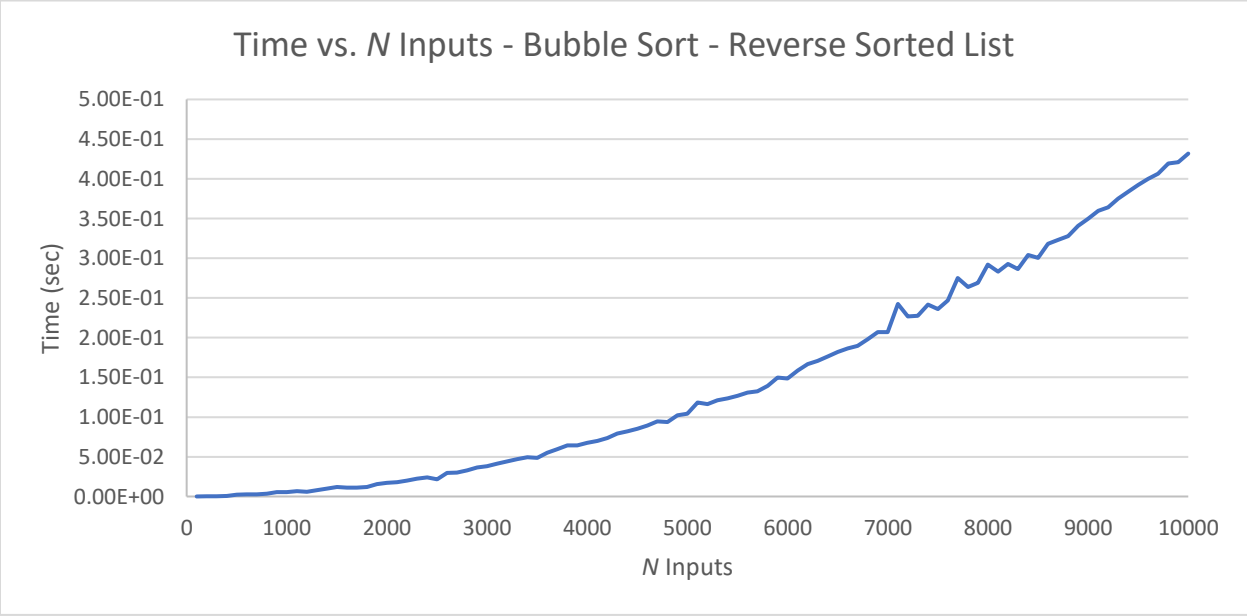
Here we see the sort that changed the most when using a different test case. Quick sort changes from a linearithmic graph to a quadratic graph, showing quick sort's worst-case scenario.

All Sorts – Previously Sorted List:



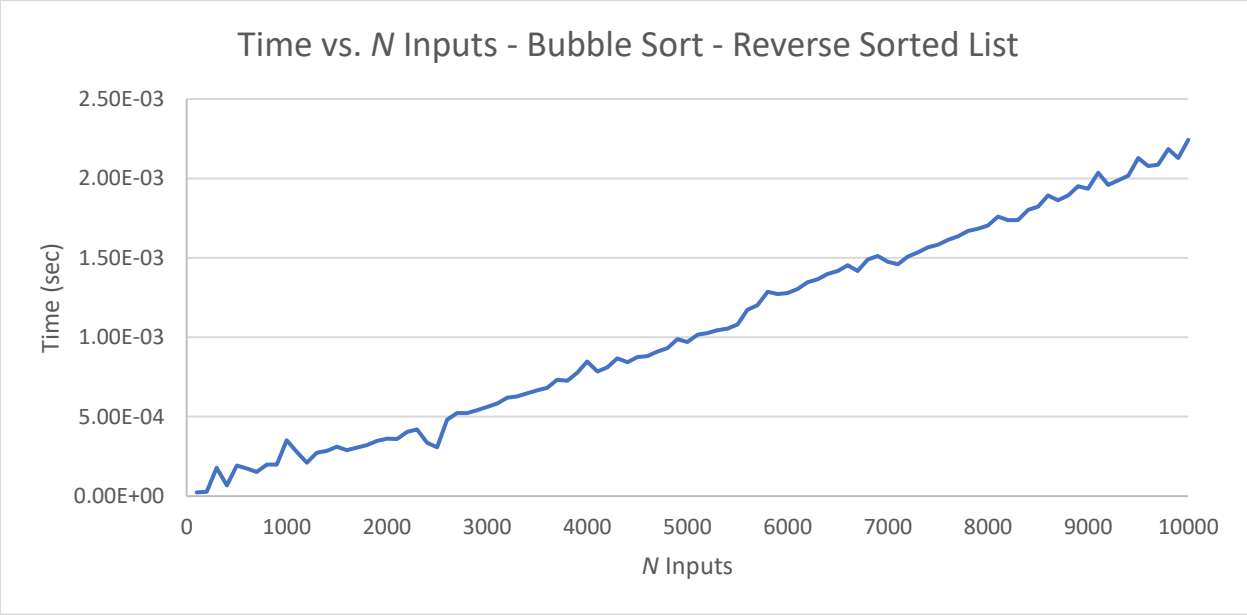
All sorts depicted here with a logarithmic y-axis to compare the different sorting performances of each algorithm with an already sorted list. Here radix sort still performs the best, however, merge sort closes the gap.

Bubble Sort – Reverse Sorted List:



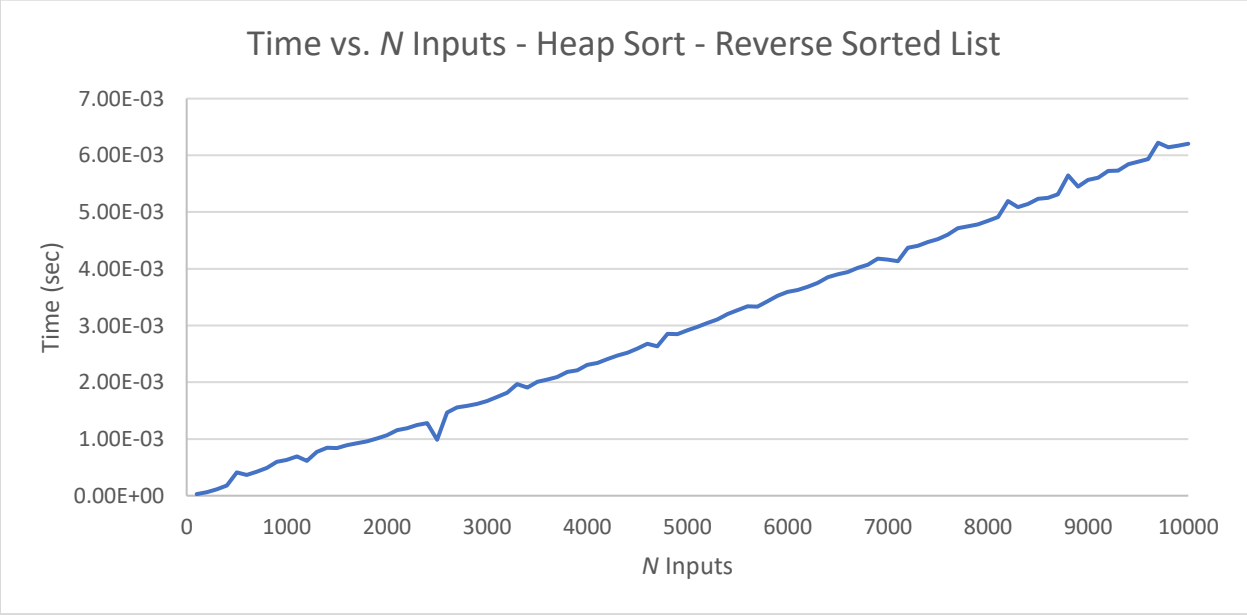
Bubble sort now takes longer than when it attempted to sort an already sorted list, however, it still outperforms itself when sorting a randomized list. Here bubble sort still performs in quadratic time.

Merge Sort – Reverse Sorted List:



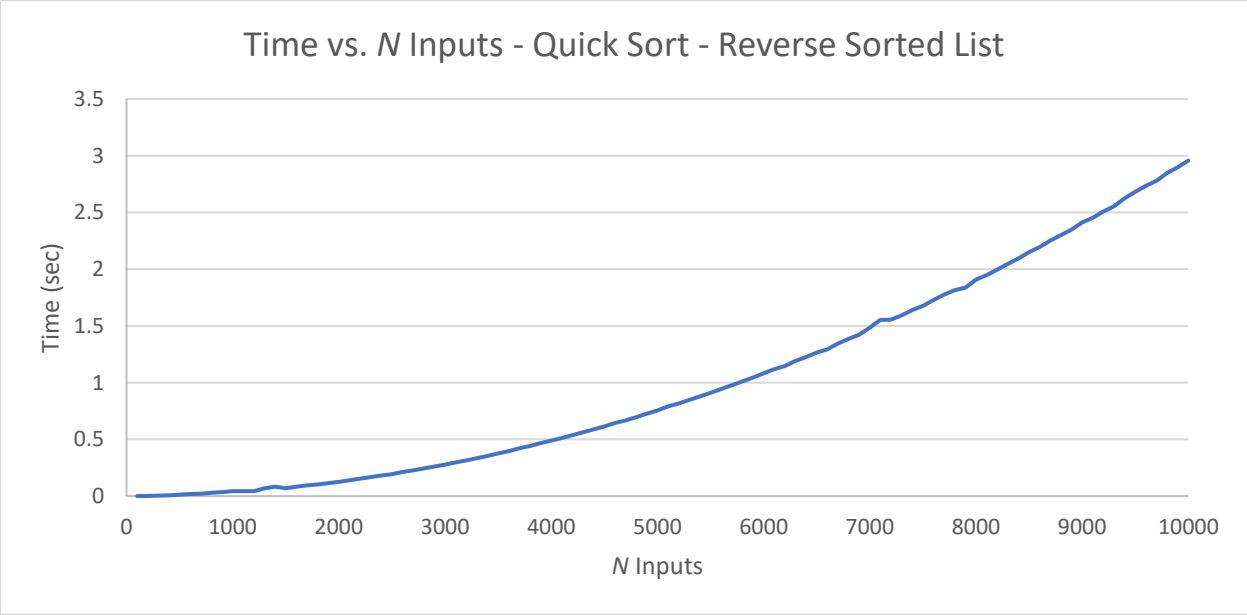
Merge Sort still performs very similarly compared to the other two test cases. Here, merge sort's graph still looks linear/linearithmic.

Heap Sort – Reverse Sorted List:



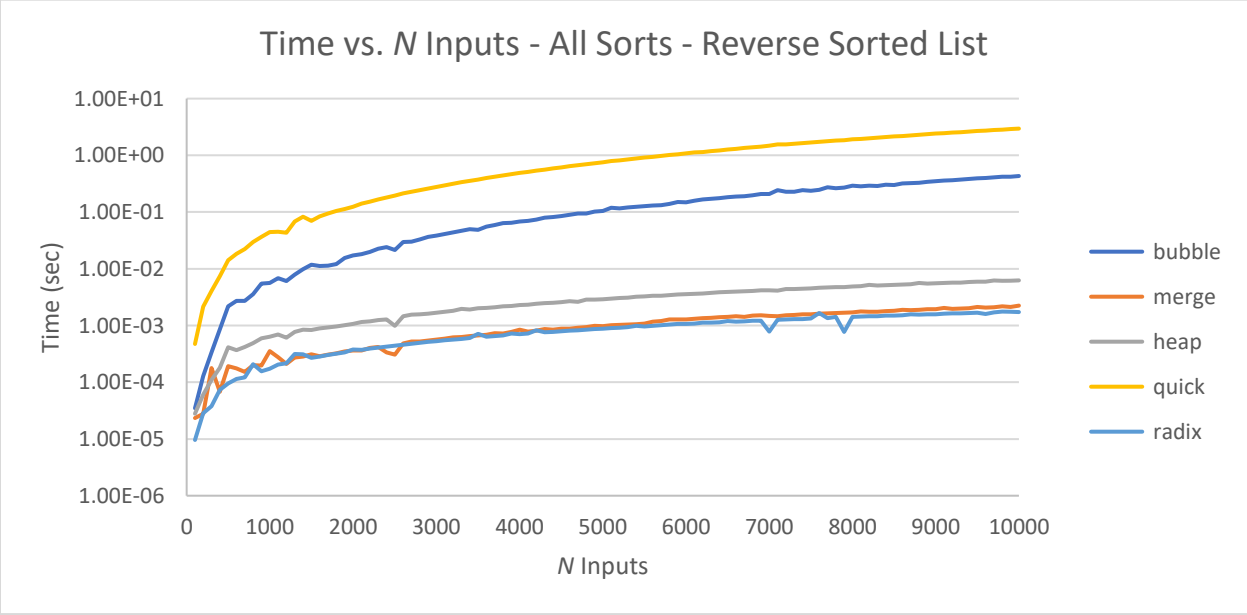
Heap sort's graph is similar to the other two test cases as well.

Quick Sort – Reverse Sorted List:



Quick sort is still performing in quadratic time which means that a reverse sorted list still forces quick sort into its worst-case scenario of $O(n^2)$.

All Sorts – Reverse Sorted List:



Radix sort still performs the best out of the other sorts while quick sort performs the worst. Merge sort and radix sort are still very close to each other.

Conclusion:

In terms of raw performance in sorting, radix sort performed the best during this project. Because of radix sort's ability to run faster than comparison sorts' lower bound of *O(nlogn)*, it allows radix sort to outperform the other 4 comparison sorts used in this project. Of the comparison sorts, with all three test cases, merge sort performs the best and has little to no discrepancies with using different test cases because of the way merge sort is implemented. Since it always divides the problem into two equal subproblems (or at least attempts to), each subproblem will require an equal amount of time to complete, allowing merge sort to run at the same time complexity when using a randomized list, an already sorted list, or even a reverse sorted list.

The most surprising (yet not that surprising at the same time) was the performance of quick sort. In the tests conducted in this experiment, quick sort was nowhere near the fastest in terms of sorting any of the lists provided. In fact, it was the worst in two of the three test cases. In the theoretical analysis, it was stated that quick sort heavily relies on a truly randomized list and the data graphs show this. With an already sorted list, both ascending and descending, quick sort performs in $O(n^2)$ time. This is because quick sort utilizes a naturally unequal partitioning of subproblems. Here, quick sort partitions based on a random pivot and uses that pivot as the divider of subproblems. However, since the quick sort utilized in this project uses the last element in the problem/list as the pivot, quick sort always divides the problem of n elements into two subproblems with one subproblem containing 0 elements, and the other containing $n-1$ elements. This does not help in the divide and conquer algorithm and rather makes quick sort run in $T(n) = n + (n-1) + (n-2) + \ldots + (n-n)$ which can be simplified to $O(n^2)$. This is the reason that makes quick sort the worst when given an already sorted list.

It is worthwhile to note here that although radix sort performs the best in these test cases, radix sort is also very inflexible in implementation. Since the test cases used in this project were lists of integers, radix sort was easily implemented as sorting through digits of an integer is trivial. However, comparison sorts such as merge sort and quick sort are far more flexible since elements can always be compared relative to each other. Radix sort relies on buckets/keys that not all objects/elements can be confined to. To further understand the complexity of radix sort, there is a report on radix sort attached to the submission of this report (also written by me). It is highly recommended to read to give further understanding to the inner workings of radix sort.