****

**PA3 Report**

By

**Andrew Vinci**

**CSI 281**

**Data Structures & Algorithms**

**Spring 2021-22**

1. **Introduction**

For this experiment, my hypothesis is that quick sort and merge sort will prove to be the most beneficial in terms of efficiency and speed. I’m basing this hypothesis on the fact that they often, in theory, hit their targets of O (n log n) and that this will prove to be more efficient as compared to some of the other sorting algorithms. As compared to bubble and insertion sort, where there is more of a chance for them to hit the marks of O (n^2), which is slower than O (n log n). I also believe that bubble sort and insertion sort will prove to be the slowest out of the 6 different algorithms, and that selection sort and shell sort will fall somewhere in between these two lower and upper bounds.

1. **Background**

**Bubble Sort:**

* A few characteristics of Bubble Sort are that large values are always sorted first.[1] Another characteristic is that it takes only one iteration of Bubble Sort in order for it to determine that the list is already sorted.[1] One of the main advantages of bubble sort is how easy it is to implement. It’s a very simple algorithm and allows for just about anyone to understand and implement it.[2] A disadvantage that is has is that it can take a very long time to run, depending on the given list size and order.[9] The performance analysis for Bubble Sort is as follows. The best-case scenario for Bubble Sort is O (n), the worst-case is O (n^2), and the average-case is O (n^2).

**Insertion Sort:**

* A characteristic of Insertion Sort is that it is adaptive, which means that is reduces its total number of steps if given a partially sorted list.[3] One of the advantages to insertion sort is that is works very well with a small list of numbers.[9] However, a disadvantage is that when introduced to a larger list, it begins to slow down and can take a long time to sort.[9] The performance analysis for Insertion Sort is as follows. The best-case scenario for Insertion Sort is O (n), the worst-case is O (n^2), and the average case is O (n^2).

**Selection Sort:**

* A few characteristics of Selection Sort is that it is a comparison-based sorting algorithm.[5] Another characteristic is that it is an “unstable” sorting algorithm, meaning that it does not preserve the order of duplicate elements.[5] It is also an in-place sorting technique, which means that the sorted items occupy the same storage as the original items.[5] One of the advantages of Selection Sort is that similar to Insertion Sort, it performs fairly well on small lists of numbers.[9] However, a disadvantage of Selection Sort is that when dealing with lists of large numbers, is isn’t able to sort them quickly, meaning that it is slow when dealing with large lists.[9] The best-case scenario for Selection Sort is O (n^2), the worst-case is O (n^2), and the average-case is O (n^2).

**Shell Sort:**

* A few characteristics of Shell sort are that it is based off of and similar to Insertion Sort.[6] Like Insertion Sort, Shell Sort is a comparison-based sorting technique and also an in-place sorting technique, which means that the items being sorted occupy the same memory as the original items.[6] One of the advantages of Shell Sort is that it works fairly well for small to mid-sized lists. One of the disadvantages of Shell Sort is that it struggles and can be slow when dealing with large lists.[9] The best-case scenario for Shell Sort is o (n), the worst-case depends on the size of the gap being applied, and the average-case also depends on the size of the gap being applied.

**Quick Sort:**

* A few characteristics of Quick Sort are that it is an in-place sorting algorithm, similar to Shell sort and Insertion Sort.[7] It is also a recursive algorithm that calls itself. Another characteristic is that it’s known as being one of the most efficient sorting algorithms when it comes to larger lists.[7] This leads into the first advantage of Quick Sort, which is that when dealing with large lists, it’s able to sort them fairly quickly and efficiently. A disadvantage to Quick Sort is that if the list is already sorted, then an algorithm such as Bubble Sort can be a more efficient algorithm to use because Bubble Sort only requires one pass through in order to check a sorted list.[9] However, it still is widely considered one of the quickest and more efficient sorting algorithms.[9] The best-case scenario for Quick Sort is O (n log n), the worst-case is O (n^2), and the average-case is O (n log n).

**Merge Sort:**

* A few characteristics of Merge Sort are that it is an efficient algorithm for sorting linked lists.[8] It is also a stable sort, which means that the same element in an array maintain their original positions.[8] An advantage of Merge Sort is that it can be used on lists of any size with good speed and efficiency.[9] A disadvantage of Merge Sort is that it requires more space than other algorithms.[9] The best-case scenario for Merge Sort is O (n log n), the worst-case is O (n log n), and the average-case is O (n log n).

1. **Implementation Detail**

**Bubble Sort:**

* Bubble Sort is a sorting algorithm that compares numbers against each other in different passes until it has finally sorted them all in either ascending or descending order, whichever is specified. In terms of characteristics, think of it as a long line of numbers. You start with the first index [0] and then compare it to the next index [1], and so on until you get to the end. As you go, you swap the index you’re currently on with the index to the right of it, based off of whether its’s smaller or larger (depending on which order you’re sorting). Once you reach the end, you start all over again in a different “pass” until the numbers are fully sorted in the specified order.

**Insertion Sort:**

* Insertion Sort is a sorting algorithm that compares numbers against a sorted list and “inserts” them into their place within the list. Again, think of a long list of numbers in a line. Start with the first index [0]. This is the start of the first list. Imagine an arbitrary line between that number and the next number, index [1]. You compare index [1] against the items on the left-side of this line, in this case index [0] and then insert the number within its place in the list based off whether it’s smaller or larger than the preceding numbers. If you’re sorting for an ascending order, then you would traverse the list until you find a number that is less than the number you’re using to search. Once you find that number, you would “insert” the number used in the search just ahead of this number. You then pick the next number in the list and continue this process until the list is sorted.

**Selection Sort:**

* Selection Sort is a sorting algorithm that does a few things. First it takes the first number of a list of numbers and then goes down the line of the list and compares that number against the current value. If you’re sorting for an ascending list, then it would go and find the smallest value in the list until it reaches the end of the list. Once it has that value, it splits the list into two separate “categories”. On the left side is the sorted list and on the right side is the unsorted list. It takes the number from the original search, in this case the lowest number, and puts it at the end of the sorted list. It then continues this process until all numbers in the unsorted list are sorted and in order in the sorted list.

**Shell Sort:**

* Shell Sort is a sorting algorithm that takes the size of the list of elements, divides that number by 2, and then uses the proceeding number to implement a gap on the list. It separates the list bases off of the gap that is given from that beginning equation. Once the gap has been set and established, it takes the index from each part of the separated numbers and compares it to the index in the same spot in the following gaps. If sorting in ascending order, it would swap the larger numbers with smaller numbers. Once it reaches the end of the comparison of gaps, the gap number is again divided by 2. The above step repeats until the gap number is 0, which occurs when dividing the gap of 1 by 2. Once this happens, bubble sort is implemented on the remaining set of numbers until the numbers are fully sorted.

**Quick Sort:**

* Quick Sort is a sorting algorithm that takes the middle element in a list of numbers, which is called the pivot, and compares against the numbers at either end of the pivot until all of the numbers on the left are less then the pivot, and all of the numbers on the right are greater than the pivot. If you’re sorting descending order, then the reverse would be true. It takes the first element on the left-hand side and then starts scanning these numbers looking for a number that is greater than the pivot. Once it finds this number, it stops. It then does something similar on the right-hand side of the pivot. It looks for a number that is less than the pivot and once it finds this number, it stops. Once both sides have stopped, they swap places in the list. This process continues until the pivot is in the right place and the numbers to the left are less than the pivot and the numbers on the right are greater than the pivot. Once again, for descending this process would be in reverse. Then the two lists, left and right of pivot, get pivots of their own and repeat this process. This process keeps repeating until the list is properly sorted, either in ascending order or descending order.

**Merge Sort:**

* Merge Sort is a sorting algorithm that uses the concept of divide and conquer in order to fully sort through a list. It first takes the list and then divides it into two halves, then it takes the two halves and divides those into two and keeps doing this until all of the elements have been divided down to the one space that they occupy. It then begins to build the list back again, but this time sorting. It takes the elements that are next to each other and puts them in a sorted pair. Then the pair is compared next to the pair that is next to those elements on the specified side of the list, remember that the list is still divided into multiple elements from the original division. Once it reaches a point where the list has been rebuilt back to the original division, it does the process all over again until the list has been sorted.

1. **Experimentation Detail**

The platform that was used to conduct this experiment is a Dell Inspiron 15 7000 series laptop that has an Intel Core i7-10510U with a CPU speed of 1.80GHz 2.30GHz. The amount of memory on board is 16gb and the operating system is a 64 bit system with a x64-based processor.

**Summary Data**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm: Bubble Sort** | | | |
| **N** | **Dataset Random** | **Dataset Ascending** | **Dataset Descending** |
| 100 | 0.0081020s | 0.0018142s | 0.0014142s |
| 100,000 | 48.363630s | 0.8231877s | 30.640366s |
| 1,000,000 | 4290.7495s = ~71.5 min | 7.7885788s | 3265.0101s = ~54.41 min |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm: Insertion Sort** | | | |
| **N** | **Dataset Random** | **Dataset Ascending** | **Dataset Descending** |
| 100 | 0.0016127s | 0.0014743s | 0.0014406s |
| 100,000 | 7.9000997s | 0.8384212s | 13.694930s |
| 1,000,000 | 705.42672s = ~11.75 min | 8.5599772s | 1274.1670 = ~21.23 min |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm: Selection Sort** | | | |
| **N** | **Dataset Random** | **Dataset Ascending** | **Dataset Descending** |
| 100 | 0.0017307s | 0.0017099s | 0.0017317s |
| 100,000 | 13.456049s | 11.706183s | 15.502362s |
| 1,000,000 | 1167.9736s = ~19.45 min | 1119.9483s = ~18.65 min | 1182.6105s = ~19.70 min |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm: Shell Sort** | | | |
| **N** | **Dataset Random** | **Dataset Ascending** | **Dataset Descending** |
| 100 | 0.0019718s | 0.0012663s | 0.0011891s |
| 100,000 | 19.986773s | 0.6569254s | 0.6579276s |
| 1,000,000 | 2035.9175s = ~33.91 min | 6.5558526s | 6.5739635s |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm: Quick Sort** | | | |
| **N** | **Dataset Random** | **Dataset Ascending** | **Dataset Descending** |
| 100 | 0.0020012s | 0.0020060s | 0.00196883s |
| 100,000 | 1.0215491s | 0.9388514s | 0.99380760s |
| 1,000,000 | 10.117513s | 9.7912063s | 9.73770956s |

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm: Merge Sort** | | | |
| **N** | **Dataset Random** | **Dataset Ascending** | **Dataset Descending** |
| 100 | 0.0024622s | 0.0026261s | 0.0015517s |
| 100,000 | 1.3349606s | 1.0762367s | 1.2353540s |
| 1,000,000 | 12.841233s | 11.238602s | 10.572838s |

1. **Discussion and Conclusion**

The results of this experiment were very interesting to watch play out. Some of my hypothesis became correct and some of the results were completely unexpected going into the experiment. In terms of the impact of the size of the lists, this experiment showcased how algorithms can become runaway trains in terms of time it takes to sort larger numbers. If you look at the averages above, you can see that for pretty much all of the algorithms, lists 100 and 100,000 were fairly efficient. The algorithm that took the longest in regard to these two lengths of lists was Bubble Sort, which took around 48 seconds to sort a random numbered list of 100,00 elements.

Once the numbers get to 1,000,000 things start to escalate. Quick Sort and Merge Sort handled this number exceptionally. In fact, it was a night and day difference between these two and some of the other algorithms. The longest that either of these two took to sort was around 12 seconds for a random array when using Merge Sort. The other algorithms had a tough time when it came to 1,000,000 elements in a list. I would say the runner up behind these two with a large list would be shell sort, however it must be noted that this is only with an ascending list or descending list. With a random numbered list, it took around 33 minutes for it to sort. With ascending and descending, it took just under 7 seconds. That’s quite the drop-off in terms of efficiency. The next in line would be Selection Sort, which averaged around 20 minutes for random, ascending, and descending lists.

In terms of consistency, Selection Sort was the most consistent outside of Quick and Merge Sort in terms of reliability of time, not necessarily pure speed though. I would still give that edge to Shell Sort, unless you need to sort a random numbered list, in which case Selection Sort would be faster by about 10 minutes. Insertion Sort was also one that varied in terms of time. For an ascending array, it took less than 10 seconds to sort, however for an ascending array it was closer to Selection Sort with 20 minutes. Out of all of these 3, Shell, Selection, and Insertion Sort it was Insertion Sort that beat out the other 3 for random numbered lists. It only took about 11 minutes to sort a random numbered list, while Shell Sort took around 30 minutes and Selection Sort took around 20 minutes.

Bubble Sort was the one, that predictably, took the longest to sort a large list. It took a little over an hour for random and a little under an hour for descending. However, with a ascending list it only took around 7 seconds. This was only bested by Shell Sort, which took around 6 seconds on average to sort an ascending array of 1,000,000.

In terms of what I expected to happen in this experiment, I would say that it did perform how I expected but it also gave me results that I had no idea would occur. For instance, when running the test, I was really surprised by how quick Bubble Sort was with ascending lists. It makes sense, because in order for Bubble Sort to validate a sorted list is only needs one pass through, but still it was quite the difference to run the algorithm for random and descending for it to take around an hour, and in between with ascending it was moving lightning fast. Also, I would say that my results backed up my hypothesis in regard to Quick Sort and Merge Sort, as these two algorithms performed the best out of the six different algorithms tested.

My ranking of the algorithms based on the results of the experiment are as follows:

1. Quick Sort
2. Merge Sort
3. Shell Sort
4. Insertion Sort
5. Selection Sort
6. Bubble Sort

The reason I put Insertion Sort above Selection Sort is because it outperformed Selection Sort when it came to a large list. However, I would give the nod to Selection Sort in terms of consistency, as it took around 20 minutes for random, ascending, and descending of the 1,000,000 list. Bubble Sort comes in last because it’s just not a super-efficient algorithm for larger lists. However, it can be incredibly useful if you just need it to quickly validate that a list is already sorted. A lot of these algorithms can also work interchangeable for smaller lists, as they all perform fairly well in regard to the smaller lists that were tested with this experiment.

If I had to pick a favorite algorithm, I would have to choose Quick Sort because of how efficient it is and how it can be used in a variety of circumstances. It is a very dependable sorting algorithm and one that I will probably choose to use in the future. I do like others though, particularly Merge Sort for a lot of the same reasons why I like Quick Sort.

1. **References**

Provide the detailed references you used for this report. You must cite all references you used throughout your report using the IEEE standard.

[1] Karuna Sehgal, “An introduction to bubble sort,” *Medium*, 12-Feb-2018. [Online]. Available: https://medium.com/karuna-sehgal/an-introduction-to-bubble-sort-d85273acfcd8. [Accessed: 25-Feb-2022].

[2] “Comparison of sorts - sorting - KS3 computer science revision - BBC Bitesize,” *BBC News*, 2022. [Online]. Available:

https://www.bbc.co.uk/bitesize/guides/z2m3b9q/revision/4#:~:text=One%20of%20the%20main%20advantages,small%20and%20simple%20computer%20program%20. [Accessed: 25-Feb-2022].

[3] J. Wandy, “The advantages & disadvantages of sorting algorithms,” *Sciencing*, 02-Mar-2019. [Online]. Available: https://sciencing.com/the-advantages-disadvantages-of-sorting-algorithms-12749529.html. [Accessed: 25-Feb-2022].

[4] Karuna Sehgal, “An introduction to insertion sort,” *Medium*, 17-Jan-2018. [Online]. Available: https://medium.com/karuna-sehgal/an-introduction-to-insertion-sort-16b97619697. [Accessed: 25-Feb-2022].

[5] “Selection sort in data structure,” *TechVidvan*, 16-Sep-2021. [Online]. Available: https://techvidvan.com/tutorials/selection-sort/. [Accessed: 25-Feb-2022].

[6] “Shell sort in data structure,” *TechVidvan*, 03-Jul-2021. [Online]. Available: https://techvidvan.com/tutorials/shell-sort/. [Accessed: 25-Feb-2022].

[7] Karuna Sehgal, “A quick explanation of Quick Sort,” *Medium*, 05-Feb-2018. [Online]. Available: https://medium.com/karuna-sehgal/a-quick-explanation-of-quick-sort-7d8e2563629b. [Accessed: 27-Feb-2022].

[8] Karuna Sehgal, “A simplified explanation of merge sort,” *Medium*, 05-Feb-2018. [Online]. Available: https://medium.com/karuna-sehgal/a-simplified-explanation-of-merge-sort-77089fe03bb2. [Accessed: 27-Feb-2022].

[9] “Advantage & disadvantages of sort,” *Advantage & Disadvantages of Sort*, 2020. [Online]. Available: http://dbmspoly.blogspot.com/p/advantage-disadvantages-of-sort.html. [Accessed: 25-Feb-2022].

1. **Appendix**

**Bubble Sort:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Bubble Sort** | | | **Dataset #: Random** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0020967 | 0.0013107 | 0.0020283 | 0.0081020s |
| 100,000 | 50.474434 | 39.305878 | 55.310595 | 48.363630s |
| 1,000,000 | 4448.6189 | 4018.4462 | 4405.1836 | 4290.7495s = ~71.5 min |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Bubble Sort** | | | **Dataset #: Ascending** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0015299 | 0.0023554 | 0.0015575 | 0.0018142s |
| 100,000 | 0.6545326 | 1.1645441 | 0.6504865 | 0.8231877s |
| 1,000,000 | 6.3105561 | 10.684835 | 6.3703453 | 7.7885788s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Bubble Sort** | | | **Dataset #: Descending** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0008699 | 0.0019166 | 0.0014563 | 0.0014142s |
| 100,000 | 26.733696 | 38.634826 | 26.552577 | 30.640366s |
| 1,000,000 | 3339.7222 | 3801.9400 | 2653.3681 | 3265.0101s = ~54.41 min |

**Insertion Sort:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Insertion Sort** | | | **Dataset #: Random** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0019714 | 0.0010439 | 0.0018229 | 0.0016127s |
| 100,000 | 8.5956392 | 5.7437465 | 9.3609142 | 7.9000997s |
| 1,000,000 | 752.43326 | 556.63464 | 807.21228 | 705.42672s = ~11.75 min |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Insertion Sort** | | | **Dataset #: Ascending** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0010302 | 0.0013386 | 0.0020541 | 0.0014743s |
| 100,000 | 0.8638308 | 0.6396160 | 1.0118168 | 0.8384212s |
| 1,000,000 | 9.0783874 | 6.6479606 | 9.9535838 | 8.5599772s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Insertion Sort** | | | **Dataset #: Descending** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0011549 | 0.0009729 | 0.0021941 | 0.0014406s |
| 100,000 | 11.555925 | 10.762067 | 18.766800 | 13.694930s |
| 1,000,000 | 1102.2446 | 1622.6456 | 1097.6109 | 1274.1670 = ~21.23 min |

**Selection Sort:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Selection Sort** | | | **Dataset #: Random** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0012465 | 0.0020115 | 0.0019342 | 0.0017307s |
| 100,000 | 13.402419 | 17.410399 | 9.5553294 | 13.456049s |
| 1,000,000 | 1066.0484 | 1444.7509 | 993.12157 | 1167.9736s = ~19.45 min |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Selection Sort** | | | **Dataset #: Ascending** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0020515 | 0.0018805 | 0.0011978 | 0.0017099s |
| 100,000 | 9.5872755 | 15.704129 | 9.8271446 | 11.706183s |
| 1,000,000 | 906.36514 | 1487.5274 | 965.95240 | 1119.9483s = ~18.65 min |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Selection Sort** | | | **Dataset #: Descending** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0021219 | 0.0021438 | 0.0009294 | 0.0017317s |
| 100,000 | 11.029692 | 18.504944 | 16.972451 | 15.502362s |
| 1,000,000 | 1124.9930 | 934.50614 | 1488.3325 | 1182.6105s = ~19.70 min |

**Shell Sort:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Shell Sort** | | | **Dataset #: Random** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0022937 | 0.0011791 | 0.0024427 | 0.0019718s |
| 100,000 | 22.505221 | 14.944350 | 22.510749 | 19.986773s |
| 1,000,000 | 2188.1174 | 1751.9852 | 2167.6499 | 2035.9175s = ~33.91 min |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Shell Sort** | | | **Dataset #: Ascending** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0011532 | 0.0014345 | 0.0012114 | 0.0012663s |
| 100,000 | 0.6422747 | 0.6511133 | 0.6773884 | 0.6569254s |
| 1,000,000 | 6.4310295 | 6.4724826 | 6.7640457 | 6.5558526s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Shell Sort** | | | **Dataset #: Descending** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0012234 | 0.0012189 | 0.0011252 | 0.0011891s |
| 100,000 | 0.6535987 | 0.6480313 | 0.6721530 | 0.6579276s |
| 1,000,000 | 6.4838464 | 6.4545767 | 6.7834675 | 6.5739635s |

**Quick Sort:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Quick Sort** | | | **Dataset #: Random** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0020301 | 0.0019761 | 0.0019974 | 0.0020012s |
| 100,000 | 1.1028427 | 0.9346853 | 1.0271195 | 1.0215491s |
| 1,000,000 | 9.7565339 | 10.115876 | 10.480132 | 10.117513s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Quick Sort** | | | **Dataset #: Ascending** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0019477 | 0.0021440 | 0.0019265 | 0.0020060s |
| 100,000 | 0.8943041 | 0.9791665 | 0.9430836 | 0.9388514s |
| 1,000,000 | 9.7010395 | 9.5546945 | 10.117885 | 9.7912063s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Quick Sort** | | | **Dataset #: Descending** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0023630 | 0.0019881 | 0.0015554 | 0.00196883s |
| 100,000 | 0.9865880 | 0.9043587 | 1.0904761 | 0.99380760s |
| 1,000,000 | 9.4631203 | 9.8299543 | 9.9200541 | 9.73770956s |

**Merge Sort:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Merge Sort** | | | **Dataset #: Random** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0033361 | 0.0018846 | 0.0021661 | 0.0024622s |
| 100,000 | 1.2336406 | 1.3868742 | 1.3843672 | 1.3349606s |
| 1,000,000 | 10.726474 | 13.318660 | 14.478567 | 12.841233s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Merge Sort** | | | **Dataset #: Ascending** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0012628 | 0.0022214 | 0.0043942 | 0.0026261s |
| 100,000 | 0.7115774 | 1.4588988 | 1.0582339 | 1.0762367s |
| 1,000,000 | 7.1830772 | 15.627264 | 10.905465 | 11.238602s |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm: Merge Sort** | | | **Dataset #: Descending** | |
| **N** | **Run #1** | **Run #2** | **Run #3** | **Average** |
| 100 | 0.0015213 | 0.0015355 | 0.0015984 | 0.0015517s |
| 100,000 | 0.7964112 | 1.8527226 | 1.0569282 | 1.2353540s |
| 1,000,000 | 7.9527707 | 14.026702 | 9.7390425 | 10.572838s |