Anthony Stem
CS 124 | Fall 2021
Project 4 Report

Dataset

This dataset is a subset of the dataset maintained by the Extrasolar Planets Encyclopaedia as of February 28, 2018. It contains data for 3732 confirmed exoplanets from various sources and missions such as Kepler, CoRoT, K2, Kelt, OGLE, and more.[1] This dataset was found on Kaggle and originally contained 97 different columns of data; however, only five columns from the original dataset were used for this project. In addition, a new column called "row" was introduced to create a unique identifier for each row in the dataset. The six fields used in the project are as follows:

• *row* - a unique integer; indicates the row in the database.
• *name* - a string; indicates the name of the exoplanet.
• *mass* - a double; represents the mass of an exoplanet (in terms of Jovian mass J_M).
• *planetaryRadius* - a double; represents the radius of the exoplanet (in terms of Jovian radii J_R).
• *orbitalPeriod* - a double; represents the orbital period (in Earth days) of the exoplanet around its host star.
• *discoveryYear* - an integer; represents the year in which the exoplanet was discovered.

The entries in this dataset are ordered numerically by *row* in ascending order.

The primary reason I chose this dataset is because I have always been fascinated by space, especially exoplanets and extrasolar objects. I wanted a dataset that would allow me to analyze and explore the various properties and attributes of different exoplanets.

Sorting Process

For this project, I measured the frequency of reads, writes, as well as the execution time for the sorting algorithms Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, Selection Sort, and Two-Sort, which, for this implementation, was a combination of Bubble Sort and Selection Sort, for vectors of sizes 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000 that contained randomized Exoplanet objects by row number. To measure the execution time for each sorting algorithm, I used the Chrono library which allowed me to create a start and stop timer before and after each sorting algorithm function call.[2] *Please note that the execution times recorded in the files may and likely will differ from*

---

[1] Original Source: http://exoplanet.eu/
Kaggle: https://www.kaggle.com/eduardowoj/exoplanets-database
[2] Chrono Library: https://www.geeksforgeeks.org/chrono-in-c/

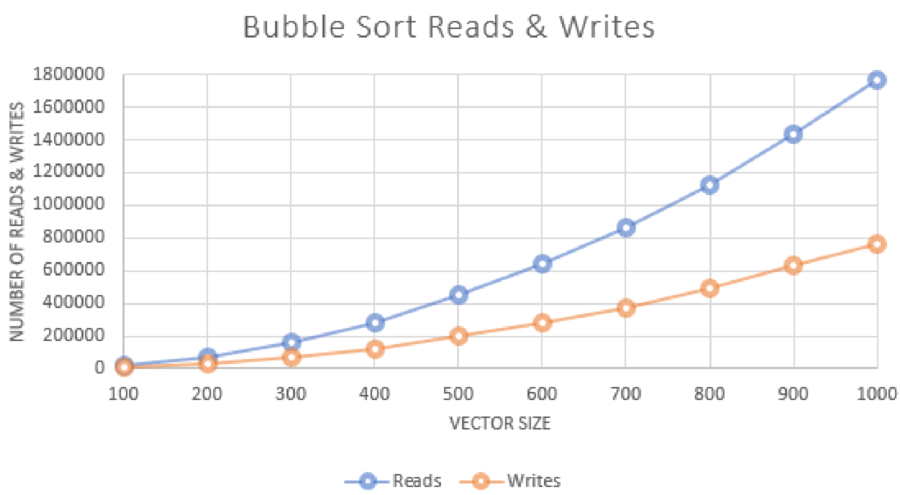
the ones listed in this report because they change after each time the program is executed.

Bubble Sort

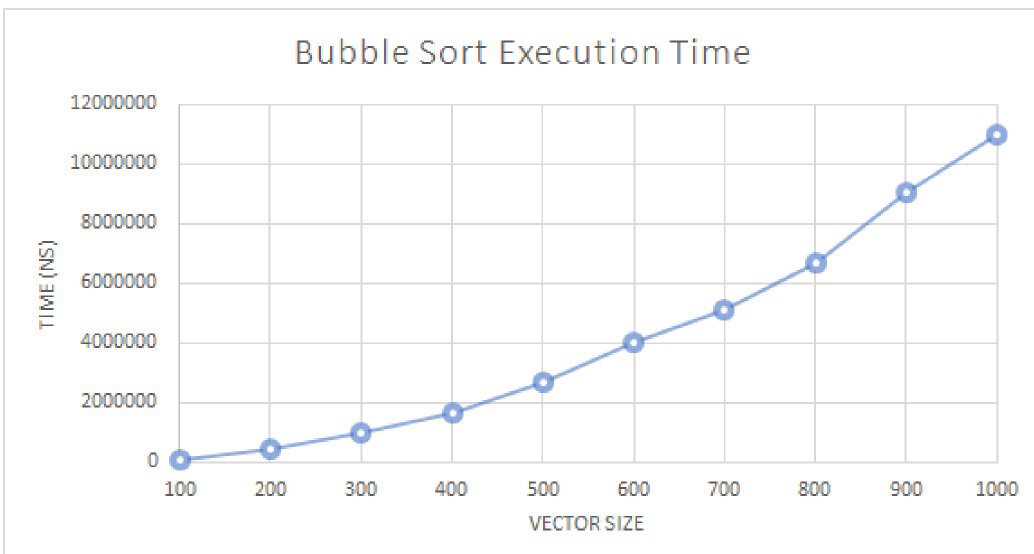
Time Complexity, Efficiency, & Stability
Bubble Sort is a *stable* sorting algorithm that has a worst-case time complexity of $O(n^2)$ and an average-case time complexity of $O(n^2)$. The auxiliary complexity for this algorithm is $O(1)$.

Results
Below is the table of reads and writes, as well as a graph containing the number of reads and writes as the vector size increases, for the Bubble Sort sorting algorithm:

Vector Size	**Reads**	**Writes**	**Run Time (μs)** ***(Extra Credit)***
100	17832	7944	116000
200	69743	30183	422600
300	161232	71544	975500
400	283189	124095	1689900
500	449150	200112	2705700
600	641022	282042	4041000
700	860388	371508	5130300
800	1125456	487248	6696900
900	1436818	629124	9063400
1000	1766211	767211	10955100



Extra Credit
The graph of function runtime in nanoseconds for each vector size for Bubble Sort is shown below:



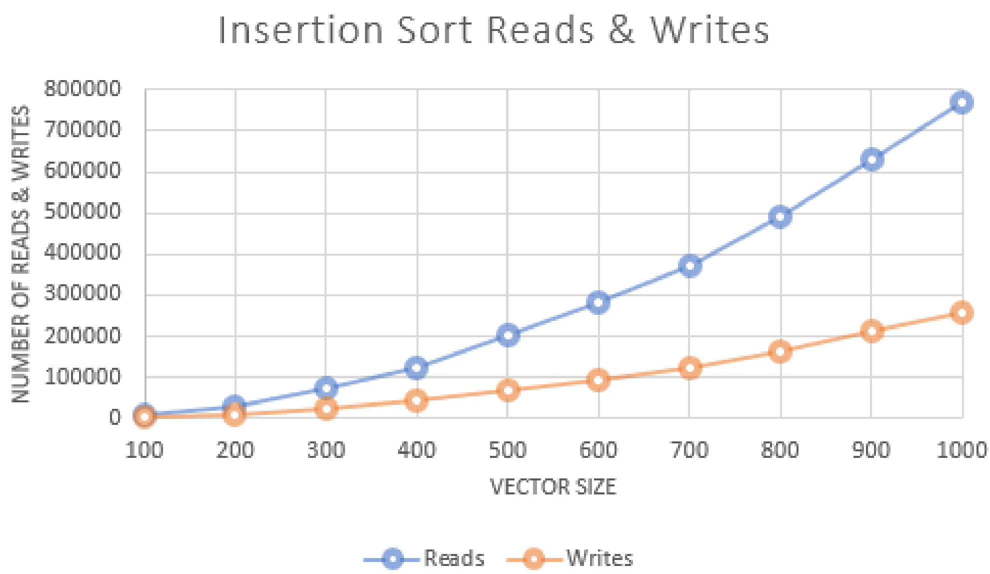
Results Interpretation
The data supports that Bubble Sort is indeed a sorting algorithm with $O(n^2)$ time complexity as demonstrated by all the graphs which follow $O(n^2)$. The number of reads is $O(n^2)$ and the number of writes is $O(n^2)$.

Insertion Sort

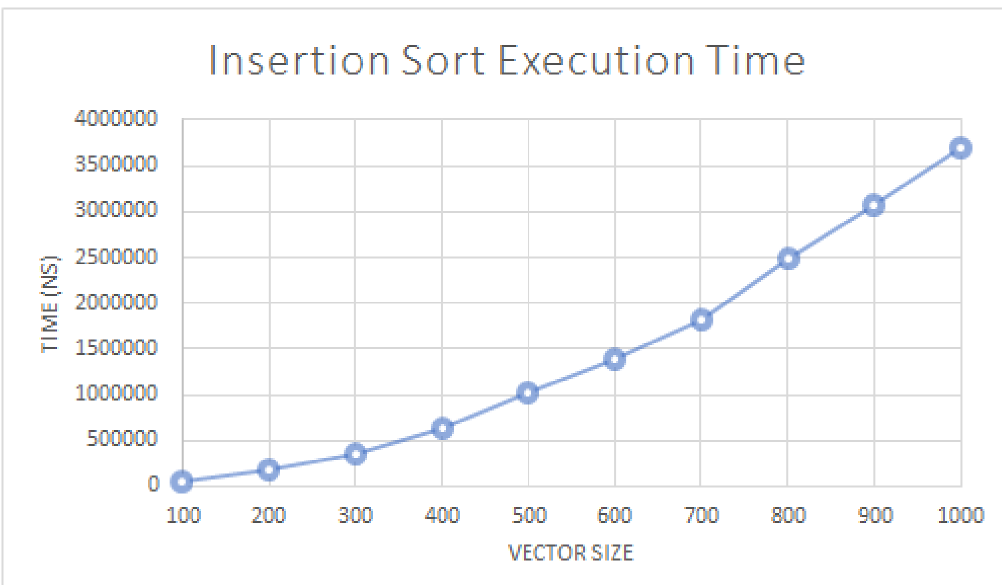
Time Complexity, Efficiency, & Stability
Insertion Sort is a *stable* sorting algorithm that has a worst-case time complexity of $O(n^2)$. The auxiliary complexity for this algorithm is $O(1)$.

Results
Below is the table of reads and writes, as well as a graph containing the number of reads and writes as the vector size increases, for the Insertion Sort sorting algorithm:

Vector Size	**Reads**	**Writes**	**Run Time (μs)** ***(Extra Credit)***
100	8142	2846	50600
200	30581	10459	171000
300	72142	24446	362900
400	124893	42163	638500
500	201110	67702	1018100
600	283240	95212	1397600
700	372906	125234	1815800
800	488846	164014	2489100
900	630922	211506	3067700
1000	769209	257735	3694200



Extra Credit
The graph of function runtime in nanoseconds and vector size for Insertion Sort is shown below:



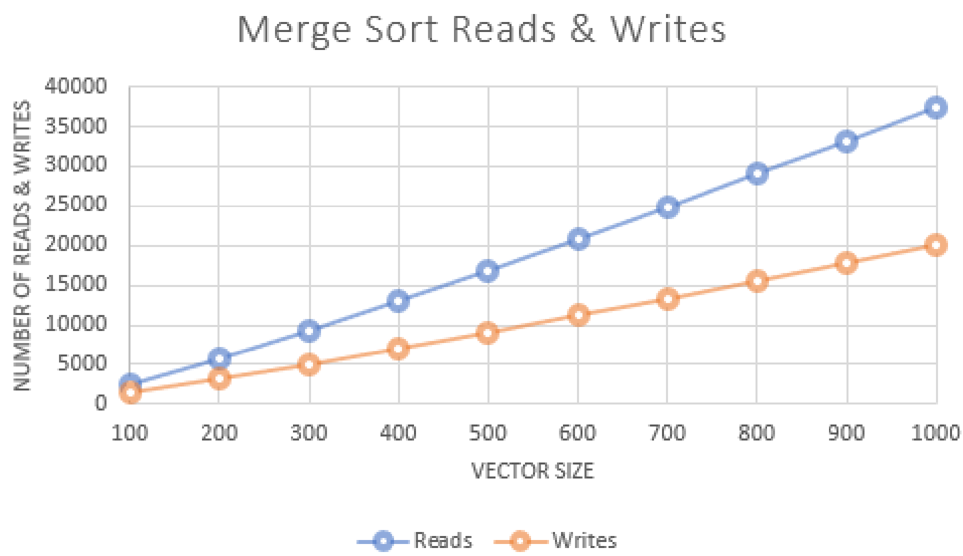
Results Interpretation
The data supports that Insertion Sort is indeed a sorting algorithm with $O(n^2)$ time complexity as demonstrated by all the graphs which follow $O(n^2)$. The number of reads is $O(n^2)$ and the number of writes is $O(n^2)$.

Merge Sort

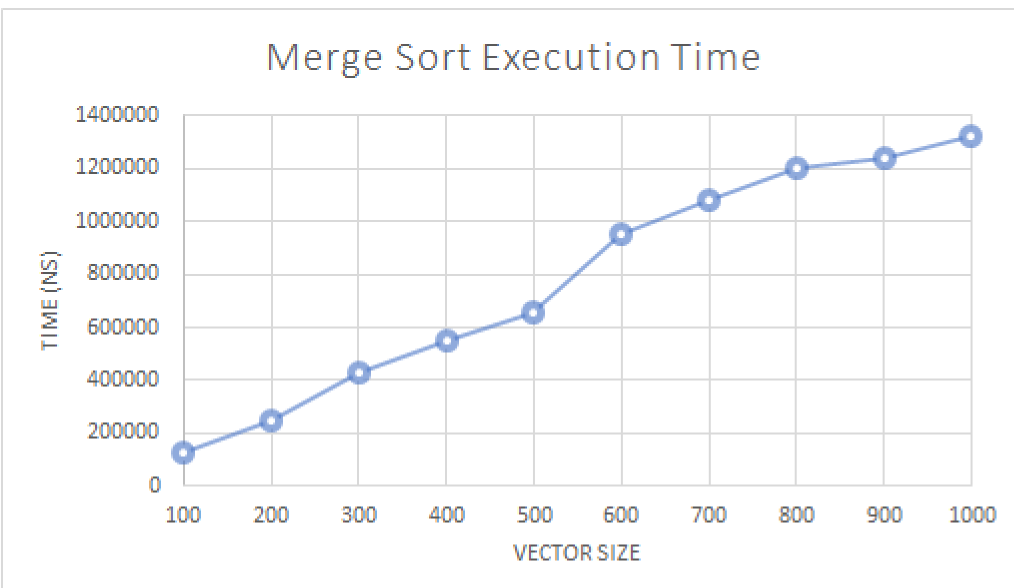
Time Complexity, Efficiency, & Stability
Merge Sort is a *stable* sorting algorithm that has a worst-case time complexity of O(nlog(n)) and an average-case complexity of O(nlog(n)). The auxiliary complexity for this algorithm is O(n).

Results
Below is the table of reads and writes, as well as a graph containing the number of reads and writes as the vector size increases, for the Merge Sort sorting algorithm:

Vector Size	Reads	Writes	Run Time (μs) *(Extra Credit)*
100	2454	1344	126300
200	5692	3088	246100
300	9166	4976	426900
400	12916	6976	549800
500	16706	8976	653100
600	20724	11152	951000
700	24832	13352	1083100
800	29010	15552	1198900
900	33190	17752	1237700
1000	37358	19952	1323400



Extra Credit
The graph of function execution time in nanoseconds for each vector size for Merge Sort is shown below:



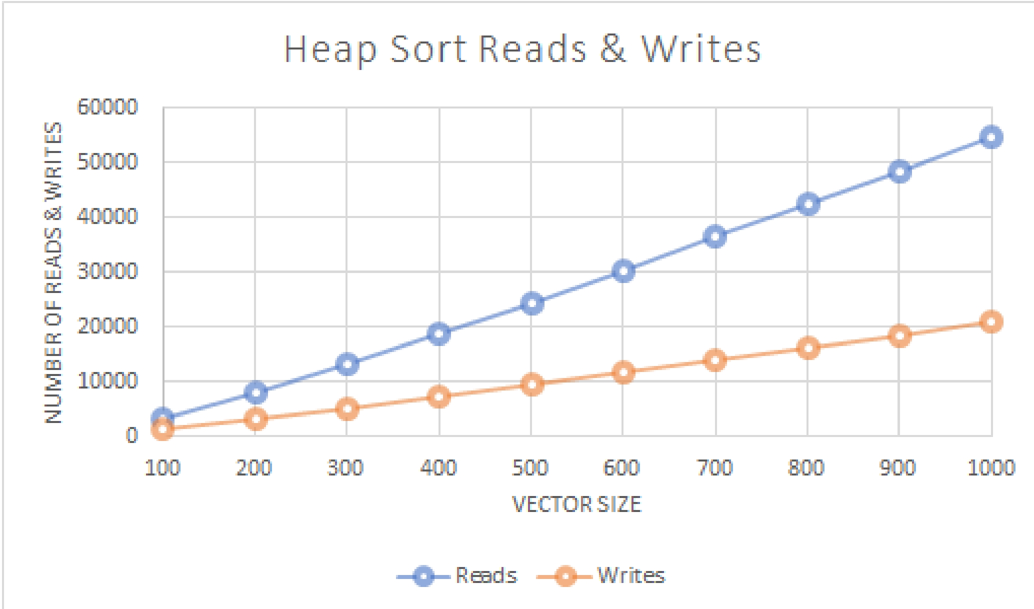
Results Interpretation
The data supports that Merge Sort is indeed a sorting algorithm with O(nlog(n)) time complexity as demonstrated by all the graphs which follow O(nlog(n)). The number of reads is O(nlog(n)) and the number of writes is O(nlog(n)).

Heap Sort

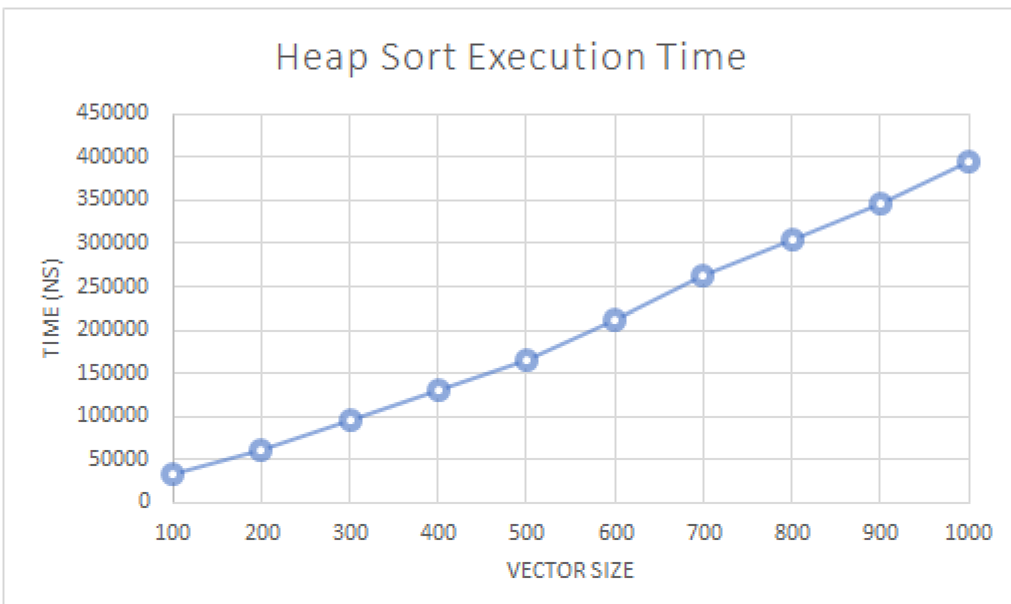
Time Complexity, Efficiency, & Stability
Heap Sort is an *unstable* sorting algorithm, which has a worst-case time complexity of O(nlog(n)) and an average-case complexity of O(nlog(n)). The auxiliary complexity for this algorithm is O(1).

Results
Below is the table of reads and writes, as well as a graph containing the number of reads and writes as the vector size increases, for the Heap Sort sorting algorithm:

Vector Size	**Reads**	**Writes**	**Run Time (μs)** ***(Extra Credit)***
100	3479	1431	32700
200	8149	3265	60800
300	13393	5289	96600
400	18797	7369	131700
500	24389	9501	166300
600	30401	11793	212500
700	36419	14075	263900
800	42416	16356	304900
900	48492	18648	346400
1000	54647	20987	395900



Extra Credit
The graph of function execution time in nanoseconds for each vector size for Heap Sort is shown below:



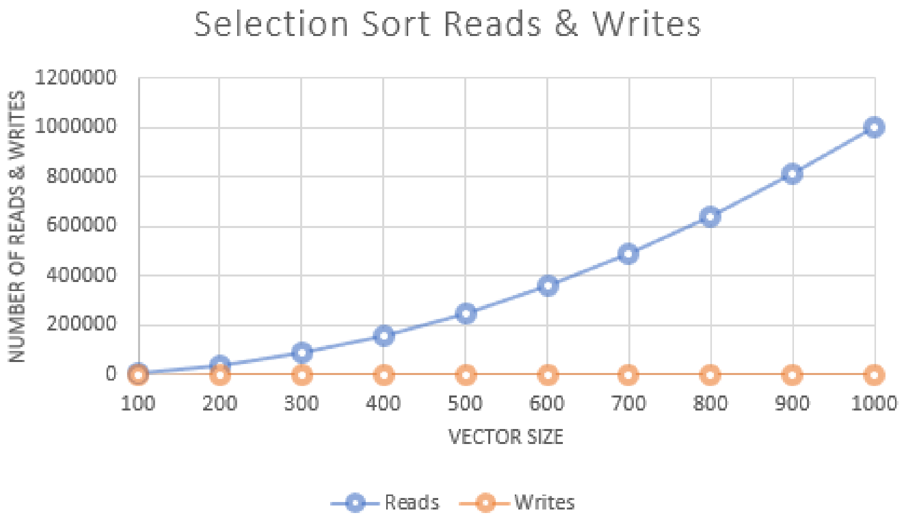
Results Interpretation
The data supports that Heap Sort is indeed a sorting algorithm with O(nlog(n)) time complexity as demonstrated by all the graphs which follow O(nlog(n)). The number of reads is O(nlog(n)) and the number of writes is O(nlog(n)).

Selection Sort *(Extra Credit)*

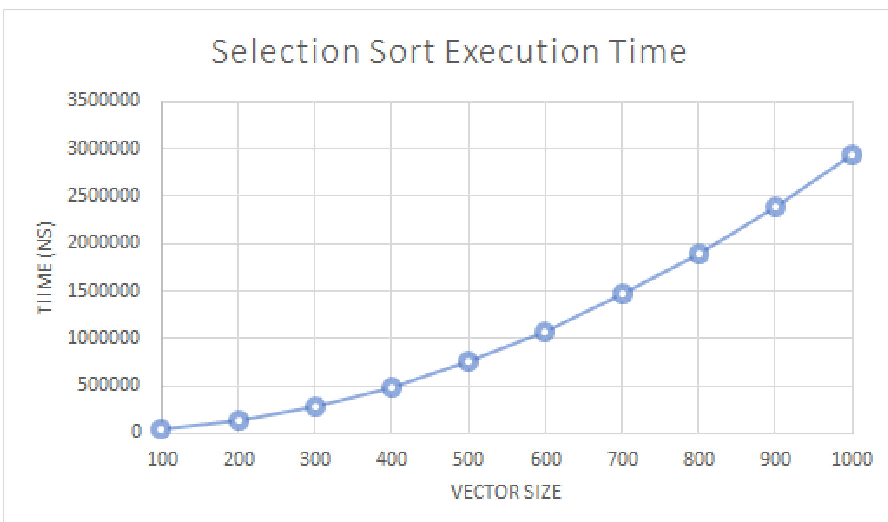
Time Complexity, Efficiency, & Stability
Selection Sort is an *unstable* sorting algorithm, which has a worst-case time complexity of $O(n^2)$ and an average-case complexity that is also $O(n^2)$. The auxiliary complexity for this algorithm is $O(1)$.

Results
Below is the table of reads and writes, as well as a graph containing the number of reads and writes as the vector size increases, for the Selection Sort sorting algorithm:

Vector Size	**Reads**	**Writes**	**Run Time (μs)** ***(Extra Credit)***
100	10197	297	36600
200	40397	597	130400
300	90597	897	278400
400	160797	1197	486000
500	250997	1497	750600
600	361197	1797	1076200
700	491397	2097	1463500
800	641597	2397	1891100
900	811797	2697	2382900
1000	1001997	2997	2930300



Extra Credit
The graph of function execution time in nanoseconds for each vector size for Selection Sort is shown below:



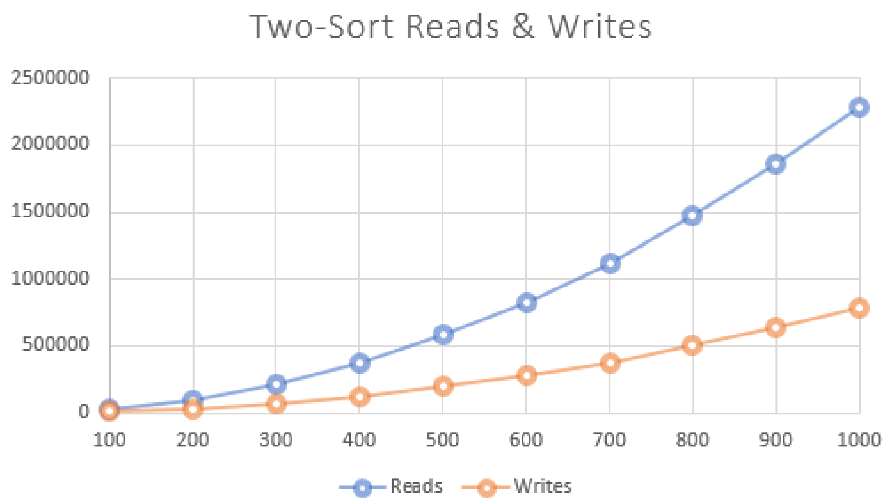
Results Interpretation
The data supports that Selection Sort is indeed a sorting algorithm with $O(n^2)$ time complexity as demonstrated by all the graphs which follow $O(n^2)$. The number of reads is $O(n^2)$; however, the number of writes or swaps is really low and actually follows $O(n)$ since we only do the swapping inside the first for-loop.

Two-Sort

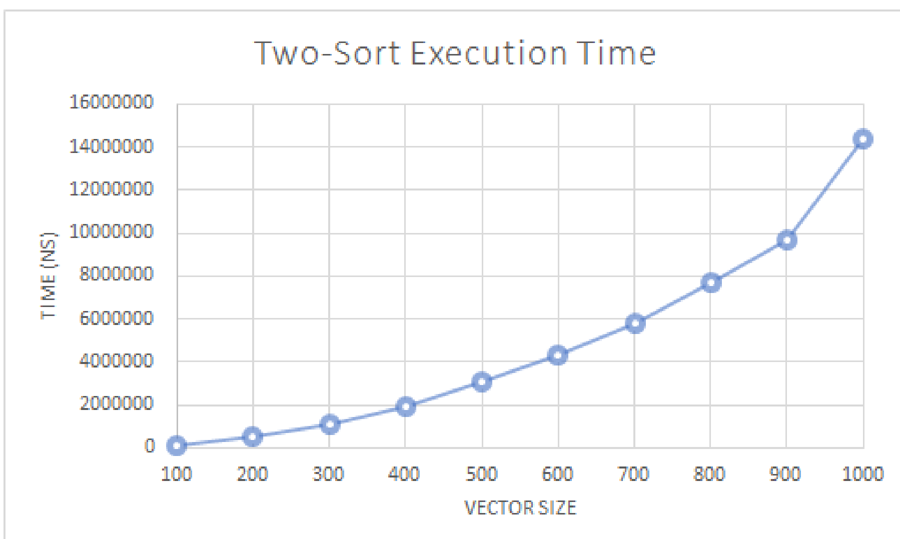
Time Complexity, Efficiency, & Stability
This two-sort implementation consists of the following two sorting algorithms: Bubble Sort and Insertion Sort. Bubble Sort is a *stable* sorting algorithm with a time complexity of $O(n^2)$, and Insertion Sort is also a *stable* sorting algorithm that also has a time complexity of $O(n^2)$. Both Bubble Sort and Insertion Sort have auxiliary complexities of O(1).

Results
Below is the table of reads and writes, as well as a graph containing the number of reads and writes as the vector size increases, for the Insertion Sort sorting algorithm:

Vector Size	Reads	Writes	Run Time (μs) ***(Extra Credit)***
100	23032	7892	138600
200	89956	30166	492700
300	208619	71333	1103600
400	368772	126748	1949800
500	581657	199399	3100400
600	826511	280273	4309100
700	1113019	376697	5777900
800	1471282	507262	7700900
900	1861243	632757	9688500
1000	2288682	778238	14382200



Extra Credit
The graph of function execution time in nanoseconds for each vector size for Heap Sort is shown below:



Results Interpretation
The data supports that Insertion is indeed a sorting algorithm with $O(n^2)$ time complexity as demonstrated by all the graphs which follow $O(n^2)$. The number of reads is $O(n^2)$ and the number of writes is $O(n^2)$.

Overall Algorithm Comparison & Analysis

Primary Sorting Algorithms
All of the individual algorithms tested—Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, and Selection Sort—have time complexities of either $O(n^2)$ or $O(n\log(n))$. Overall the slowest sorting algorithms of the ones tested, which had time complexities of $O(n^2)$, are Bubble Sort, Insertion Sort, and Selection Sort, and the fastest sorting algorithms of the ones that were tested, which had time complexities of $O(n\log(n))$, are Merge Sort and Heap Sort

Of the algorithms with time complexities of $O(n^2)$, Insertion Sort had the least amount of reads for each vector size for all vector sizes. Selection Sort had the least amount of writes for all vector sizes, and, interestingly, Selection Sort is the only algorithm out of the ones tested where the number of reads increased linearly. Selection Sort also had the shortest execution time for each vector for all the vector sizes tested. Conversely, Bubble Sort had the most amount of reads out of all the $O(n^2)$ sorting algorithms. In addition, it also had the most number of writes and the largest execution time out of all the $O(n^2)$ algorithms.

For the algorithms with time complexities of $O(n\log(n))$, Merge Sort had the least number of reads and writes compared to other algorithms with $O(n\log(n))$ time complexities for all vector sizes; however, it did have a greater execution time compared to Heap Sort. On the other hand, Heap Sort had the most reads and writes, but the fastest execution time compared to all $O(n\log(n))$ algorithms tested.

Out of all algorithms tested, Bubble Sort was the least efficient as it had the most number of reads, writes, as well as the greatest execution times for all tests. The most efficient algorithm based on the number of reads was Merge Sort, and, surprisingly, the most efficient algorithm based on writes was Selection Sort because the number of writes only increases by $O(n)$ instead of the expected $O(n^2)$; also, $O(n)$ is more efficient than the other algorithms with $O(n\log(n))$, which is why this algorithm was much more efficient than the others in terms of the number of writes it had to make to sort the vector each time. The most efficient algorithms in terms of execution time were were Heap Sort and Merge Sort with both of them having time complexities of $O(n\log(n))$. The least efficient algorithms in terms of execution time were Insertion Sort and Bubble Sort which both had time complexities of $O(n^2)$. Overall, the following table summarizes the overall efficiency rankings of each sorting algorithm for reads, writes, and execution time based on the data collected for all of the vector sizes that were tested:

Efficiency Ranking	By Reads	By Writes	By Execution Time
1. Best	Merge Sort	Selection Sort	Heap Sort
2	Heap Sort	Merge Sort	Merge Sort
3	Insertion Sort	Heap Sort	Selection Sort
4	Selection Sort	Insertion Sort	Insertion Sort
5. Worst	Bubble Sort	Bubble Sort	Bubble Sort

Note: The colors are there just to indicate which sorting algorithm is which for readability.

Two-Sort
Another algorithm that was tested was two-sort, which was implemented using Bubble Sort and Selection Sort. First, the algorithm would use Insertion Sort to sort the vector of exoplanets based on the row number. Next, the algorithm would use Bubble Sort, a stable sorting algorithm, to sort the vector of exoplanets based on the discovery year of each exoplanet. The resulting sorted vector would contain exoplanets that are sorted based on their discovery year, but all of the exoplanets with the same discovery year would be ordered by their row number. An example of such behavior is shown in the following output of a the first 6 elements after Bubble Sort and after Insertion Sort:

```
------------------------
1       OGLE-2016-BLG-1469L b           13.6        -1          -1              2017
2       11 Com b                        19.4        -1      326.03              2008
7       16 Cyg B b                      1.68        -1       799.5              1996
8       18 Del b                        10.3        -1       993.3              2008
22      2M 2140+16 b                      20      0.92        7340              2010
24      2M 2236+4751 b                  12.5        -1          -1              2016
-----
3461    PSR 1257 12 c                  0.013        -1     66.5419              1992
186     GJ 229 B                          35        -1          -1              1995
7       16 Cyg B b                      1.68        -1       799.5              1996
3470    PSR J2051-0827 b                28.3        -1   0.0991103              1996
3723    tau Boo b                       5.84      1.06     3.31249              1996
56      Aldebaran b                     6.47        -1      628.96              1998
------------------------
```

Overall, this Two-Sort implementation was quite slow compared to all of the other sorting algorithms tested, considering it has a time complexity of $O(n^2)$ and takes a lot of reads and writes, as well as time to make a complete sort; however, this is to be expected since we are actually sorting by two different algorithms. The benefit of a Two-Sort implementation, though, is you can sort a data based on two attributes. For future implementations, however, one might consider choosing more efficient algorithms to sort their data than the ones that were chosen for this test.

Overview & Conclusion

All in all, these tests demonstrate that each of the sorting algorithms we analyzed have their own strengths and weaknesses, warranting their use in certain use cases or scenarios. For example, consider the following questions/usage cases:

1. If you need to sort a contacts list on a mobile app, which sorting algorithm(s) would you use and why?
Since mobile devices tend to have limited computational power, we should take into consideration memory consumption while still considering time efficiency. However, contact lists do tend to be relatively small in size, so we do not have to worry much about really long execution times for large list sizes. In addition, when sorting a contact list on a mobile app, a user is likely to sort by *First Name* and *Last Name* for all their contacts, so which may have duplicate first names or last names, so a stable sorting algorithm may be in order for this task. Good candidates for such task are ones that minimize reads and writes, and, overall, consume little memory (have a low auxiliary complexity). Therefore, some decent candidates are Selection Sort, which uses very little writes (uses only O(n) writes) and a relatively small amount of reads, and Merge Sort, which doesn't use a lot of reads or writes (they are both O(nlog(n)). However, Selection Sort is better than Merge Sort for small amounts of data (e.g. a contact list), so, depending on the size of the contact lists expected for this mobile app, a one might consider using either Selection Sort (for smaller list sizes) or Merge Sort (for larger list sizes).

2. What about if you need to sort a database of 20 million client files that are stored in a data center in the cloud?
To sort a database containing 20 million client files, a fast and efficient sorting algorithm would be optimal for this case. Also, because the data is stored on the cloud, we can assume that the database's computational power is strong and reliable. Therefore, we could choose a fast sorting algorithm without really having to worry much about memory consumption. A good candidate for this task is Merge Sort, which has a fast time complexity of O(nlog(n)) but a relatively high auxiliary complexity of O(n). As stated before, the hardware is likely able to handle such memory consumption, thereby making it a good potential choice to quickly sort all 20 million client files. Another good choice would be Quick Sort, which is an algorithm not discussed in this report but is a fast sorting algorithm with an average case time complexity of O(nlog(n)).