CMPS 2120

Prof Morales

Stuart Wurtman

Kyle Sandridge

# Search & Sort Algorithm Report

Finding algorithms that efficiently search and sort data is necessary to manipulate large amounts of raw data. A list containing 10 or 100 elements is relatively fast to search through and sort, but if the list grows to contain 10000, 100000, or more elements, then the time it takes to search through and sort the list grows too. The growth rate of the algorithm determines how the algorithm behaves as the input increases. For instance, algorithms with factorial growth become intractable after a relatively small increase in input. The aim is to keep search and sort time low, thus we must identify algorithms whose time complexity grows the least. Efficient algorithms allow us to quickly search and sort through large data sets and provide a tool to create programs and solve problems with efficiency.

In this project, the algorithms are categorized into searching algorithms and sorting algorithms. The search algorithms that were used in the project are binary search, linear search, and interpolation search. The sorting algorithms used in the project are selection sort, heap sort, and quick sort. This project was designed as an experiment to determine which search algorithm and which sorting algorithm are the most efficient and how the computing environment affects results.

For search algorithms, the hypothesis was that the interpolation search algorithm would be quickest. Interpolating data in mathematics usually means trying to obtain more accurate data with less error, so “interpolation search” reminds us of this definition. Further, with a uniformly distributed dataset, interpolation search can have O(log log n) avg time complexity, whereas the binary search is O(log n) and linear search is O(n). All three algorithms have a constant best case, O(1). However, linear

search and interpolation search have a worst case of O(n), whereas binary search has a worst case of O(log n).

| Theoretical complexity of the search algorithms | | | | |
|---|---|---|---|---|
| | Time Complexity | | | Space Complexity |
| **Algorithm** | **Best Case** | **Avg Case** | **Worst Case** | **Auxiliary Space** |
| **Linear Search** | O(1) | O(n) | O(n) | O(1) |
| **Binary Search** | O(1) | O(log n) | O(log n) | O(1) |
| **Interpolation** | O(1) | O(log log n) or O(n) depending | O(n) | O(1) |

Figure 1.

From the data collected, it was evident that the binary search was the quickest search algorithm. However, interpolation search was nearly as quick. Given that the average time complexity of interpolation search varies between O(log log n) and O(n), it was interesting that interpolation search performed similarly to binary search, in approximately O(log n) time, which is evident from the horizontal line seen in figure 3. The linear search, as the name suggests, had linear results, which made it the worst performing algorithm.

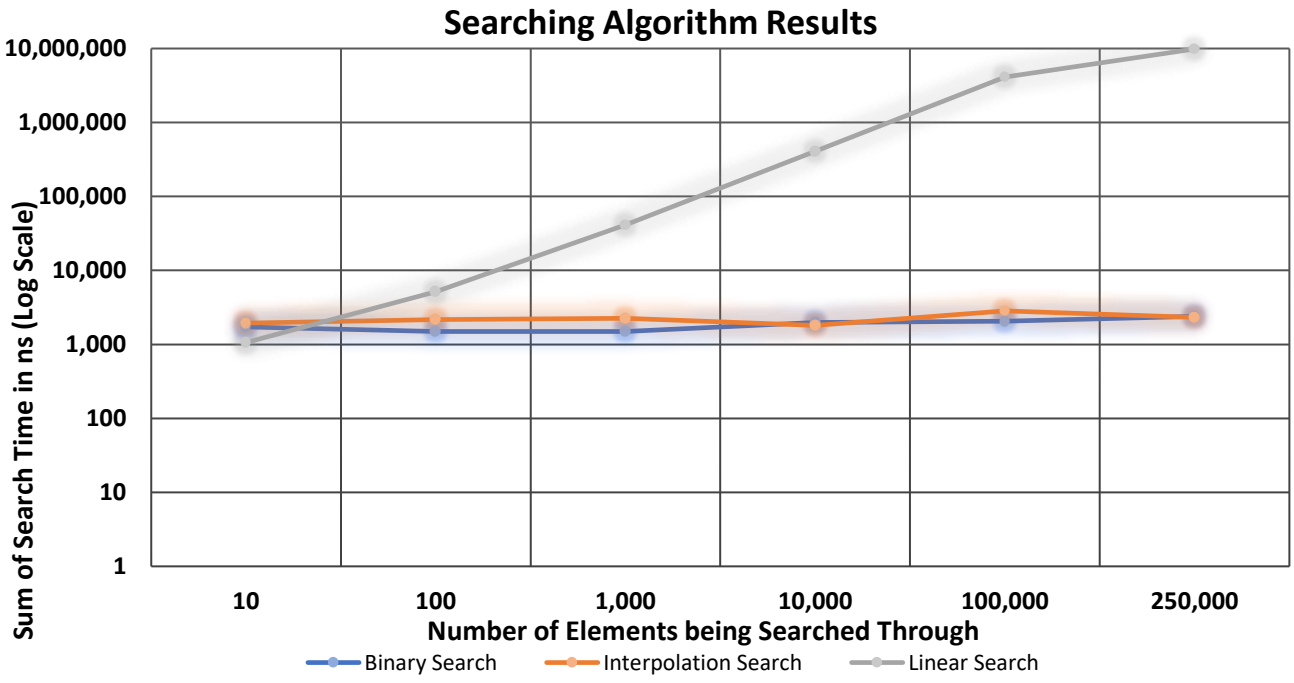


Figure 2.

Sorting a dataset is more computationally intensive than searching through an ordered dataset for a single element. In this project, we implemented three sorting algorithms: heap sort, quick sort, and selection sort.

For sorting algorithms, the hypothesis was that the quick sort algorithm would be quickest, as the name suggests. The average time complexities of quick sort and heap sort are both O(n log n), but quick sort has a best case of O(n) compared to heap sort which is O(n log n). Quick sort has a worst case of $O(n^2)$ compared to heap sort which is O(n log n). Selection sort is $O(n^2)$ in best, average, and worst cases, therefore it should perform worse. Whether quick sort would be the fastest sorting algorithm was not clear until tested.

| **Theoretical complexity of the sorting algorithms** | | | | |
|---|---|---|---|---|
| | **Time Complexity** | | | **Space Complexity** |
| **Algorithm** | **Best Case** | **Avg Case** | **Worst Case** | **Auxiliary Space** |
| **Selection Sort** | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | O(1) |
| **Heap Sort** | O(n log n) | O(n log n) | O(n log n) | O(1) |
| **Quick Sort** | O(n) | O(n log n) | $O(n^2)$ | O(n) or O(log n) depending |

Figure 3.

After the results of the experiment were collect, quick sort was faster than heap sort, but when plotted (figure 4), quick sort and heap sort run nearly parallel, indicating the same growth rate. This reflects their shared average case time complexity, O(n log n). The quick sort algorithm was the fastest of the three sorting algorithms. The $O(n^2)$ worst case for quick sort is a specific case that should not be triggered by a uniformly distributed dataset, and it is evident that we did not encounter a worst case in our experiment.

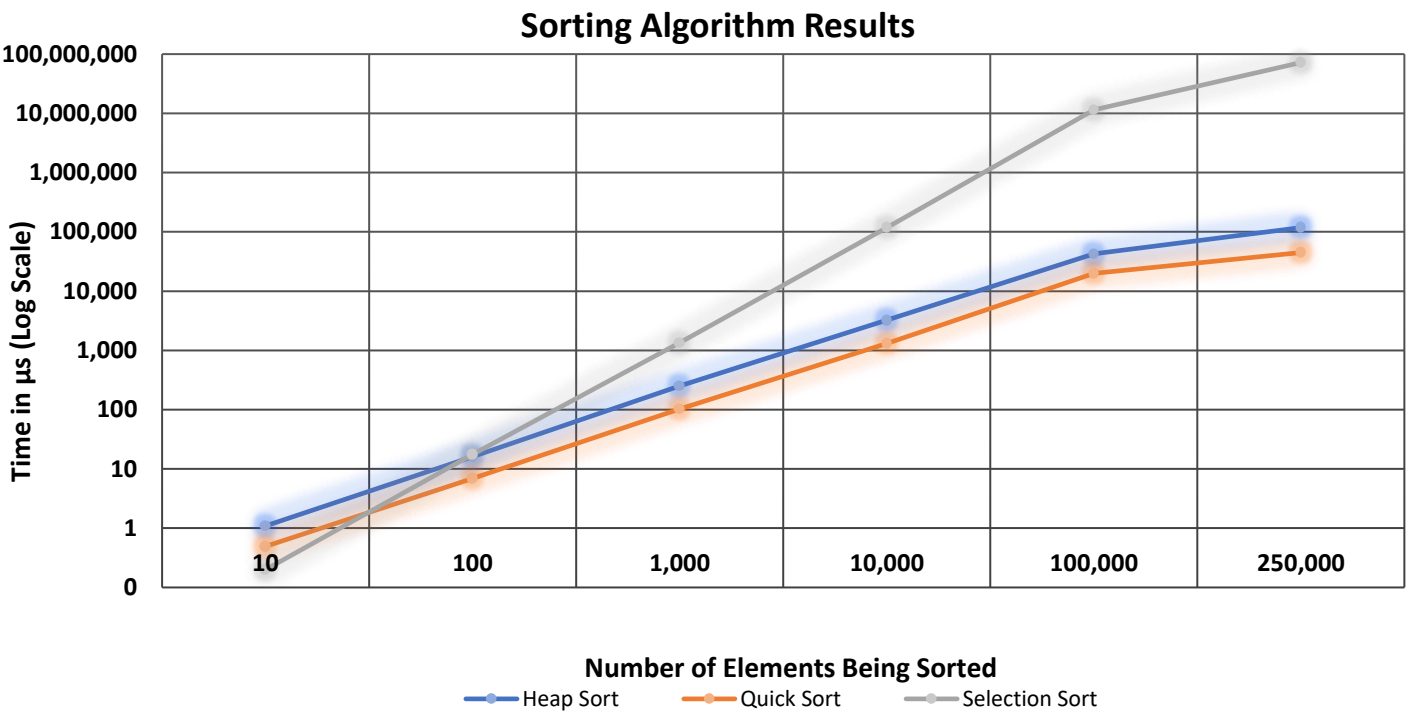


Figure 4.

After analyzing the data and corresponding graphs, we must reject our first hypothesis, that the interpolation search algorithm would perform the fastest of the three search algorithms, because the results of this experiment were that the binary search algorithm performed the fastest. For sorting algorithms, we accept our hypothesis as true, the quick sort algorithm performed the fastest in this experiment.

As part of this project, we ran each of our tests five times to ensure we had accurate data. All the tests were run back-to-back by a program which wrote the results to two csv files: searchResults.csv and sortResults.csv. The program, input files, and result files can be found on GitHub at https://github.com/m4ngo5/2120-Project

We labored to create an environment free of confounding variables. Two environments were tested. First, a Linux virtual machine was created on the Microsoft Azure cloud as a test bench. Creating a VM with dedicated hardware resources allows the tests to be conducted without interference from

other applications or processes. However, the results indicated that the VM was returned timestamps with no greater than 100ns resolution. There was not time to fully understand or correct this problem, but one possible explanation is that when the program requests a timestamp from the operating system (Debian), it returns the time known to it, which in turn would be the time provided by the VM. This layer of abstraction may result in the timestamp being rounded to the nearest 100ns.

The second environment we tested provided the results for the experiment. The computer used in our tests was a 2018 Razer Blade Stealth (laptop) with 16GBs of DDR4 ram, and a quad core Intel i7-8550U processor clocked at 1.8gHz. The clock speed is significant to the results, because the program that runs the computations executes on only a single logical processor (a quad core hyperthreaded processor has 8 logical processors). This fact was observed first during tests on the Linux VM. The Linux VM had 4 cores and the program execution resulted in a CPU load of 25%, which is 100% utilization of 1 core (figure 5).

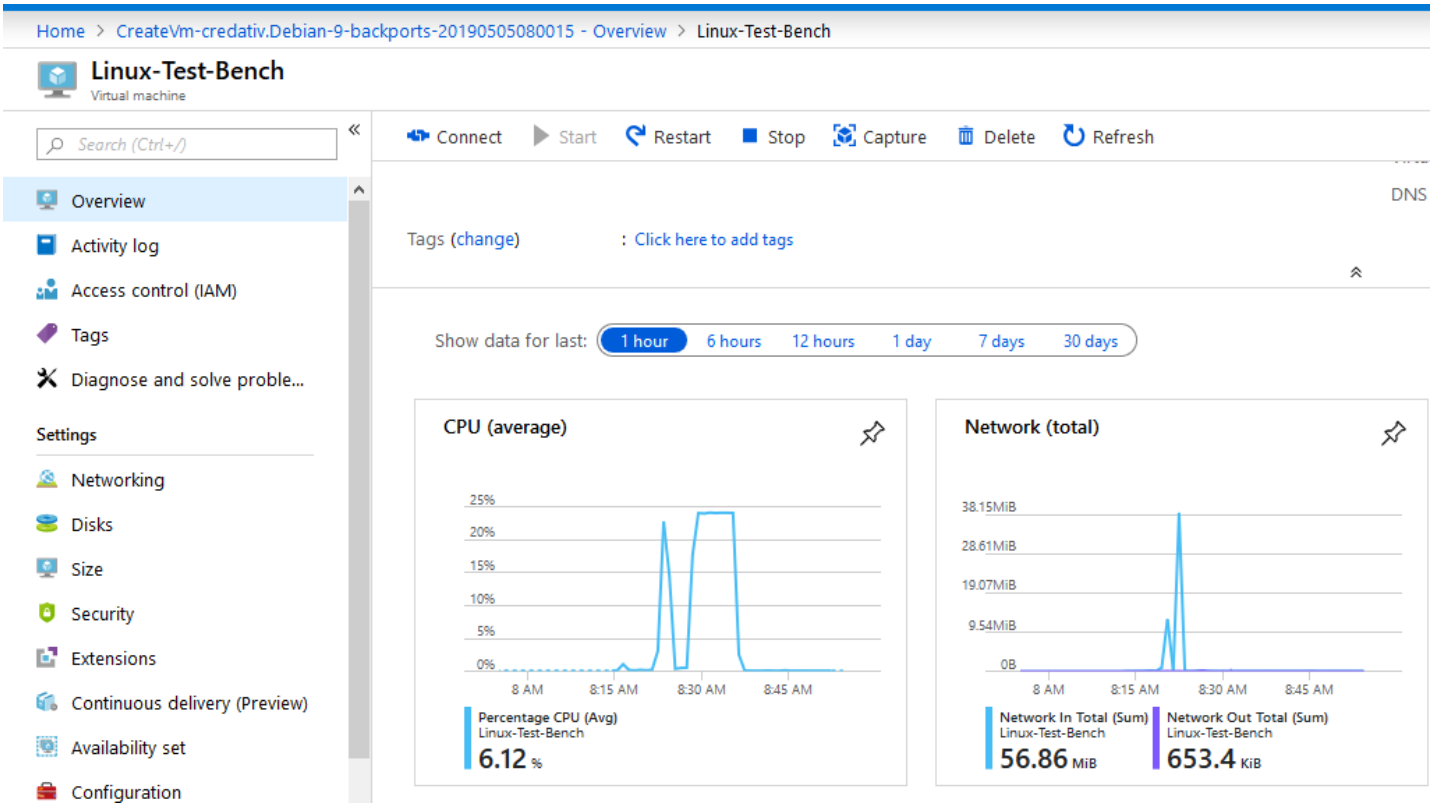


Figure 5. (The first CPU spike is the search algorithms; the second CPU spike is the sorting algorithms.)

While the results of the experiment are straight-forward, they provide space for further experiments and investigation. Performing the experiment on the Linux VM would allow greater control of the environment, if another method of high-resolution time measurement could be found that works in the VM. The quick sort implementation used in this experiment was a pure quick sort implementation that allowed full recursion down to the single element; a hybrid quick sort implementation could work more quickly. Tail recursion was not implemented in quicksort, which would reduce call stack frames; tail recursion is intended to reduce auxiliary space requirements and it is unknown to the authors whether tail recursion will increase or decrease execution time. The authors were previously unfamiliar with the heap sort algorithm and optimizations to heap sort are unknown but worth investigating. Finally, we remark that the experiment deserves repeating with various datasets on a consistent hardware configuration. Which datasets could give us O(log log n) results for interpolation search, and which datasets could give us $O(n^2)$ results for quick sort?

The following pages contains the source code of the following files in order:

- searchSortMain.cpp
- searchAlgos.h
- sortAlgos.h
- searchAglos.cpp
- sortAglos.cpp

Find the full set of project files along with the presentation and report on GitHub.

https://github.com/m4ngo5/2120-Project