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.

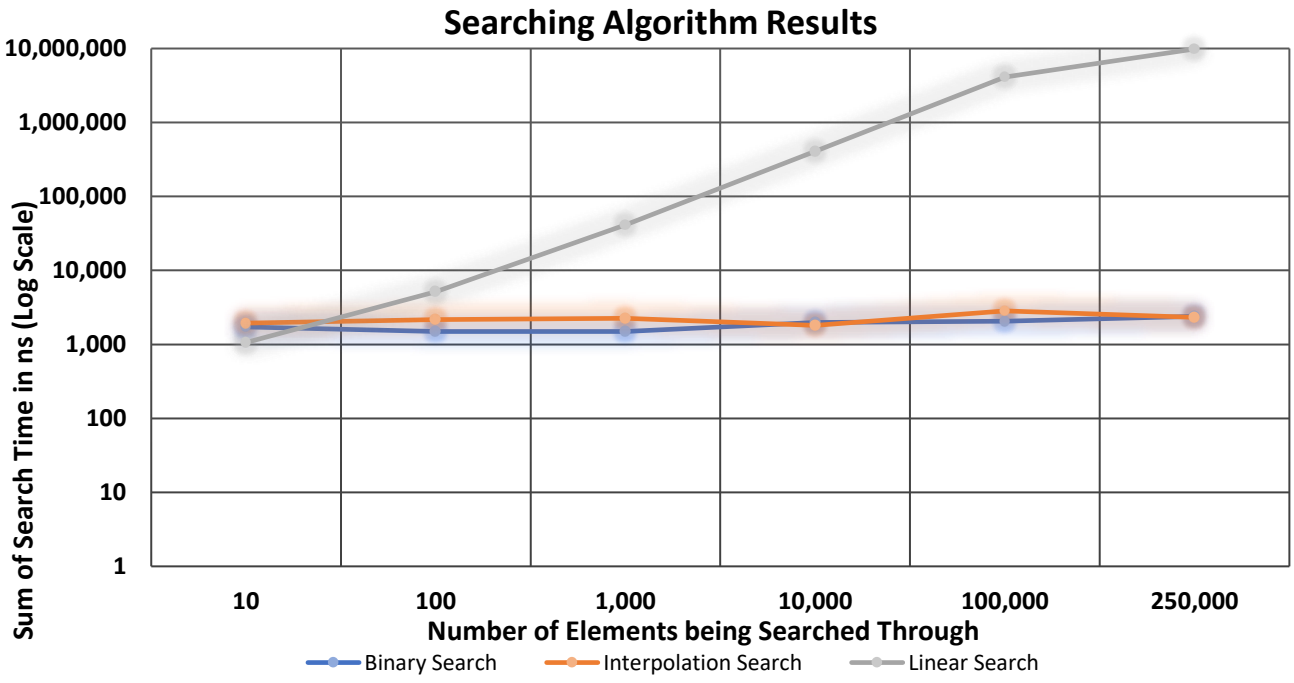

Figure 2.

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.

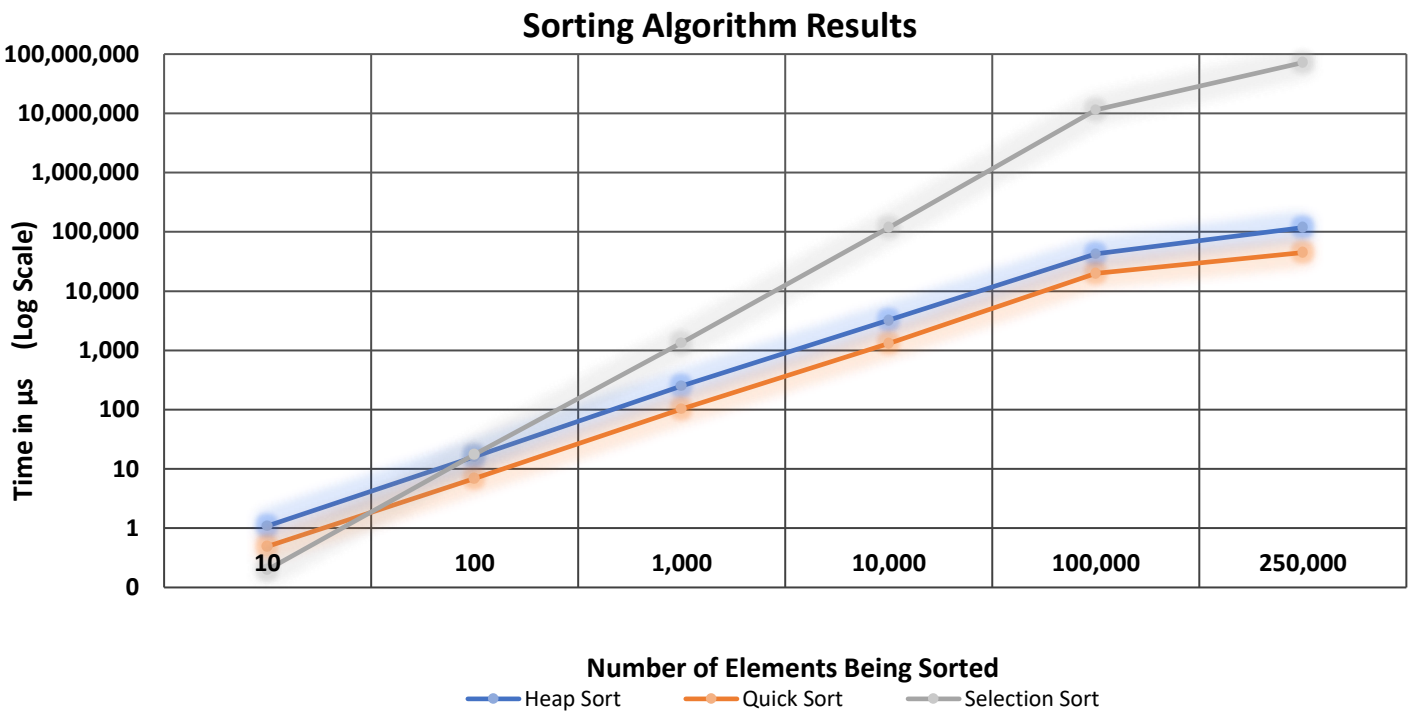

Figure 4.

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).

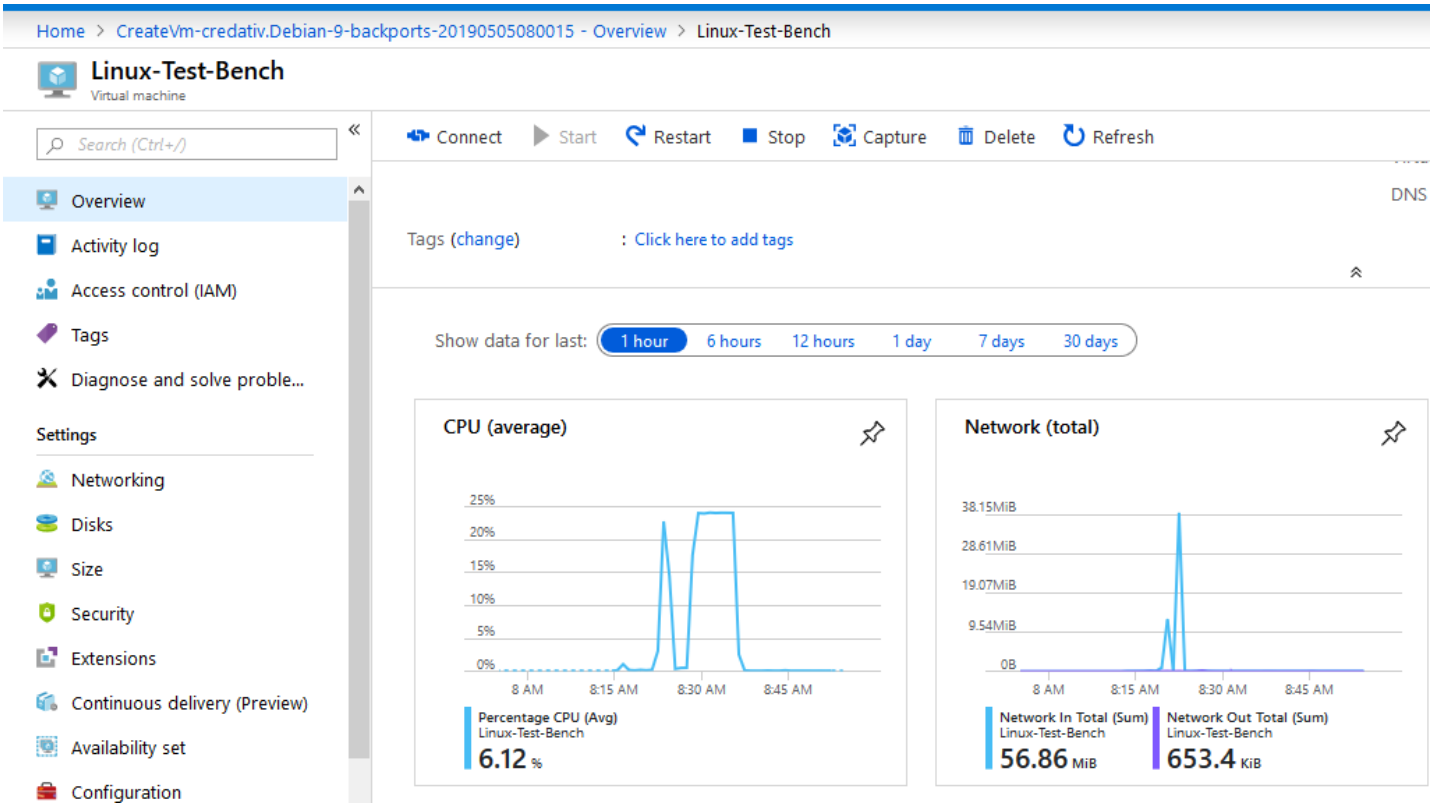

Figure 5. (The first CPU spike is the search algorithms; the second CPU spike is the sorting algorithms.)

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

```cpp
// searchSortMain.cpp
// This program executes 3 search algorithms and 3 sorting algorithms and measures their time complexity using clocks and counters.

// The three search algorithms are:
// 1 - Binary Search
// 2 - Linear Search
// 3 - Interpolation Search

// The three sorting algorithms are:
// 1 - Selection Sort
// 2 - Heap Sort
// 3 - Quick Sort

// The search functions will repeat each test 'x' times according to the value of the defined constant 'REPS'.
// The time calculations do not seem to work when compiled for Windows.  There is probably a missing system dependency.
#include <chrono>
#define TIME_UNIT nanoseconds
#define TIME_UNIT2 microseconds
#define SEARCH_REPS 5
#define SORT_REPS 5
using namespace std;
using namespace std::chrono;
// Global Variable Declarations
// Declare start and stop time_point variables
high_resolution_clock::time_point start;
high_resolution_clock::time_point stop;

// Declare nanosecond and microsecond timestamp variables
typedef std::chrono::duration<int,std::nano> timestamp;
timestamp time1, time2, time3, time4, time5, time6, avg;
typedef std::chrono::duration<int,std::micro> timestamp2;
timestamp2 time1micro, time2micro, time3micro, time4micro, time5micro, time6micro;

// Counter variable
int count = 0;

// Include language libraries
#include <iostream>
#include <fstream>
#include <cctype>

// Include the search and sort algorithm header files
#include "sortAlgos.h"
#include "searchAlgos.h"

using namespace std;
using namespace std::chrono;

int main(){

    // Instantiate search algorithm object
    searchAlgos startSearches;
    // Run searches a specified number of times
    startSearches.runSearches(SEARCH_REPS);

    // Instantiate sort algorithm object
    sortAlgos startSort;
    // Run sorts a specified number of times
    startSort.sortAll(SORT_REPS);

    // End program
    return 0;
}

```

```cpp
// searchAlgos.cpp
// 1 - Binary Search
// 2 - Linear Search
// 3 - Interpolation Search

#ifndef SEARCHALGOS_H
#define SEARCHALGOS_H

#include <iostream>
#include <fstream>
#include <cctype>
#include <chrono>
using namespace std;
using namespace std::chrono;

class searchAlgos{
    private:
        int primesArr[250000];
        void loadPrimes(int loadSize);
        int binarySearch(int arr[], int arrSize, int elem);             //returns count of iterations
        int linearSearch(int arr[], int arrSize, int elem);             //returns count of iterations
        int interpolationSearch(int arr[], int arrSize, int elem);      //returns count of iterations
    public:
        searchAlgos();
        ~searchAlgos();
        void runSearches(int repetitions);
        void binary(int repetitions, int arrSize, int a, int b, int c);
        void linear(int repetitions, int arrSize, int a, int b, int c);
        void interpolation(int repetitions, int arrSize, int a, int b, int c);
};

#include "searchAlgos.cpp"
#endif
```

```cpp
// sortAlgos.cpp
// 1 - Selection Sort
// 2 - Heap Sort
// 3 - Quick Sort

#ifndef SORTALGOS_H
#define SORTALGOS_H

#include <iostream>
#include <fstream>
#include <cctype>
#include <chrono>
using namespace std;
using namespace std::chrono;

class sortAlgos{
    private:
        int sort10[10];
        int sort100[100];
        int sort1000[1000];
        int sort10000[10000];
        int sort100000[100000];
        int sort250000[250000];
        void loadArray(int* arr, int cap, const std::string& source);
        int selectionSort(int arr[], int arrSize);
        int heapSort(int arr[], int arrSize);
        int quickSort(int arr[], int low, int high);
        void heapify(int arr[], int heapSize, int root);
        int partition(int arr[], int low, int high);
        void flip(int* a, int* b);
        void printArray(int arr[], int arrSize); // for debugging and testing correct implementation of each sorting algorithm
    public:
        sortAlgos();
        ~sortAlgos();
        void shuffleArrays();
        void sortAll(int repetitions);
        void selectionSorting(ofstream&);
        void heapSorting(ofstream&);
        void quickSorting(ofstream&);
};
#include "sortAlgos.cpp"
#endif
```

```cpp
// sortAlgos.cpp
// 1 - Selection Sort
// 2 - Heap Sort
// 3 - Quick Sort

// The search functions return the count of search iterations, rather than the position of a found element.
// The search functions return -1 if the element is not found.

searchAlgos::searchAlgos(){
}
searchAlgos::~searchAlgos(){
}
void searchAlgos::runSearches(int repetitions){
    // Load primes numbers from primes.txt into primesArr
    loadPrimes(250000);

    // Create output filestream
    ofstream ofs;

    // Open output filestream object to searchResults.csv and overwrite any existing data
    ofs.open("searchResults.csv");

    // Write column headers to searchResults.csv
    ofs << "Algorithm,Array Size,Repetition,Elem 1,Elem 1 Time (ns),Elem 1 Iterations,Elem 2,Elem 2 Time (ns),Elem 2 Iterations,Elem 3,Elem 3 Time (ns),Elem 3 Iterations,Avg (ns)" << endl;
    ofs.close();

    // Binary Search Algorithm Searches
    // usage: binarySearchThree(primesToSearch, searchPrime1, searchPrime2, searchPrime3)
    binary(repetitions, 10, 2, 5, 7);
    binary(repetitions, 100, 3, 229, 523);
    binary(repetitions, 1000, 3, 3571, 7907);
    binary(repetitions, 10000, 3, 48611, 104723);
    binary(repetitions, 100000, 3, 611953, 1299689);
    binary(repetitions, 250000, 3, 1655131, 3497849);

    // Linear Search Algorithm Searches
    // usage: linear(repetitions, primesToSearch, searchPrime1, searchPrime2, searchPrime3)
    linear(repetitions, 10, 2, 5, 7);
    linear(repetitions, 100, 3, 229, 523);
    linear(repetitions, 1000, 3, 3571, 7907);
    linear(repetitions, 10000, 3, 48611, 104723);
    linear(repetitions, 100000, 3, 611953, 1299689);
    linear(repetitions, 250000, 3, 1655131, 3497849);

    // Interpolation Search Algorithm Searches
    // usage: binarySearchThree(primesToSearch, searchPrime1, searchPrime2, searchPrime3)
    interpolation(repetitions, 10, 2, 5, 7);
    interpolation(repetitions, 100, 3, 229, 523);
    interpolation(repetitions, 1000, 3, 3571, 7907);
    interpolation(repetitions, 10000, 3, 48611, 104723);
    interpolation(repetitions, 100000, 3, 611953, 1299689);
    interpolation(repetitions, 250000, 3, 1655131, 3497849);
}
void searchAlgos::binary(int repetitions, int arrSize, int a, int b, int c){
    // Write algorithm run information to csv
    ofstream ofs;
    ofs.open("searchResults.csv", std::ofstream::app);
    for(int i = 0; i < repetitions; i++){
        ofs << "Binary Search," << arrSize << "," << i+1 << ",";

        //search a
        count = binarySearch(primesArr, arrSize, a);
        time1 = duration_cast<TIME_UNIT>(stop - start);
        ofs << a << "," << time1.count() << "," << count << ",";

```

```
66          //search b
67          count = binarySearch(primesArr, arrSize, b);
68          time2 = duration_cast<TIME_UNIT>(stop - start);
69          ofs << b << "," << time2.count() << "," << count << ",";
70
71          //search c
72          count = binarySearch(primesArr, arrSize, c);
73          time3 = duration_cast<TIME_UNIT>(stop - start);
74          ofs << c << "," << time3.count() << "," << count << ",";
75
76          //average
77          ofs << 1.0*(time1.count() + time2.count() + time3.count())/3 << endl;
78      }
79      ofs.close();
80  }
81  int searchAlgos::binarySearch(int arr[], int arrSize, int elem){
82      start = chrono::high_resolution_clock::now();
83      int left = 0, right = arrSize-1, mid;
84      count = 1;
85      while(left <= right){
86          mid = left + (right - 1)/2; //integer division takes the floor
87          if(elem > arr[mid]) left = mid + 1;
88          else if(elem < arr[mid]) right = mid - 1;
89          else if(elem == arr[mid]){
90              stop = chrono::high_resolution_clock::now();
91              return count;
92          }
93          count++;
94      }
95      stop = chrono::high_resolution_clock::now();
96      return -1;
97  }
98  void searchAlgos::linear(int repetitions, int arrSize, int a, int b, int c){
99      ofstream ofs;
100     ofs.open("searchResults.csv", std::ofstream::app);
101     for(int i = 0; i < repetitions; i++){
102         //Write algorithm run information to csv
103         ofs << "Linear Search," << arrSize << "," << i+1 << ",";
104
105         //search a
106         count = linearSearch(primesArr, arrSize, a);
107         time1 = duration_cast<TIME_UNIT>(stop - start);
108         ofs << a << "," <<
            duration_cast<nanoseconds>(stop-start).count()/*time1.count()*/ << "," << count
            << ",";
109
110         //search b
111         count = linearSearch(primesArr, arrSize, b);
112         time2 = duration_cast<TIME_UNIT>(stop - start);
113         ofs << b << "," << time2.count() << "," << count << ",";
114
115         //search c
116         count = linearSearch(primesArr, arrSize, c);
117         time3 = duration_cast<TIME_UNIT>(stop - start);
118         ofs << c << "," << time3.count() << "," << count << ",";
119
120         //average
121         ofs << 1.0*(time1.count() + time2.count() + time3.count())/3 << endl;
122     }
123     ofs.close();
124 }
125 int searchAlgos::linearSearch(int arr[], int arrSize, int elem){
126     start = chrono::high_resolution_clock::now();
127     int i;
128     for(i = 0; i < arrSize; i++){
129         if(arr[i] == elem){
130             stop = chrono::high_resolution_clock::now();
131             return i+1;
132         }
```

```
133         }
134         stop = chrono::high_resolution_clock::now();
135         return -1;
136     }
137     void searchAlgos::interpolation(int repetitions, int arrSize, int a, int b, int c){
138         //Write algorithm run information to csv
139         ofstream ofs;
140         ofs.open("searchResults.csv", std::ofstream::app);
141         for(int i = 0; i < repetitions; i++){
142             ofs << "Interpolation Search," << arrSize << "," << i+1 << ",";
143 
144             //search a
145             count = interpolationSearch(primesArr, arrSize, a);
146             time1 = duration_cast<TIME_UNIT>(stop - start);
147             ofs << a << "," << duration_cast<nanoseconds>(stop-start).count() << "," <<
                count << ",";
148 
149             //search b
150             count = interpolationSearch(primesArr, arrSize, b);
151             time2 = duration_cast<TIME_UNIT>(stop - start);
152             ofs << b << "," << time2.count() << "," << count << ",";
153 
154             //search c
155             count = interpolationSearch(primesArr, arrSize, c);
156             time3 = duration_cast<TIME_UNIT>(stop - start);
157             ofs << c << "," << time3.count() << "," << count << ",";
158 
159             //average
160             ofs << 1.0*(time1.count() + time2.count() + time3.count())/3 << endl;
161         }
162         ofs.close();
163     }
164     int searchAlgos::interpolationSearch(int arr[], int arrSize, int elem){
165         start = chrono::high_resolution_clock::now();
166         int left = 0, right = arrSize-1, pos;
167         count = 1;
168         while(left <= right && elem >= arr[left] && elem <= arr[right]){
169             if(left == right){
170                 if(arr[left] == elem){
171                     stop = chrono::high_resolution_clock::now();
172                     return count;
173                 }
174                 stop = chrono::high_resolution_clock::now();
175                 return -1;
176             }
177             pos = left + ( ( (double)(right - left) / (arr[right] - arr[left])) * (elem -
                arr[left]));
178             if(arr[pos] == elem){
179                 stop = chrono::high_resolution_clock::now();
180                 return count;
181             }
182             if(arr[pos] < elem) left = pos + 1;
183             else{
184                 right = pos - 1;
185             }
186             count++;
187         }
188         stop = chrono::high_resolution_clock::now();
189         return -1;
190     }
191     void searchAlgos::loadPrimes(int loadSize) {
192         //Open filestream to primes.txt and read first 250000 primes into primesArr
193         ifstream fin;
194         fin.open("primes.txt");
195         for(int i = 0; i < loadSize; i++){
196             fin >> primesArr[i];
197         }
198         fin.close();
199     }
```

```cpp
// sortAlgos.cpp
// 1 - Selection Sort
// 2 - Heap Sort
// 3 - Quick Sort


sortAlgos::sortAlgos(){
}
sortAlgos::~sortAlgos(){
}
void sortAlgos::shuffleArrays(){
    loadArray(sort10, 10, "List10.txt");
    loadArray(sort100, 100, "List100.txt");
    loadArray(sort1000, 1000, "List1000.txt");
    loadArray(sort10000, 10000, "List10000.txt");
    loadArray(sort100000, 100000, "List100000.txt");
    loadArray(sort250000, 250000, "List250000.txt");
}
void sortAlgos::loadArray(int* arr, int cap, const std::string& source){
    ifstream ifs;
    ifs.open(source);
    for(int i = 0; i < cap; i++){
        ifs >> arr[i];
    }
}
void sortAlgos::sortAll(int repetitions){
    ofstream ofs;
    ofs.open("sortResults.csv");
    ofs << "Algorithm,Repetition,List10.txt Time(ns),List10.txt Iterations,List100.txt Time(ns),List100.txt Iterations,List1000.txt Time(ns),List1000.txt Iterations,List10000.txt Times(ns),List10000.txt Iterations,List100000.txt Time(ns),List100000.txt Iterations,List250000.txt Times(ns),List250000.txt Iterations,Avg Time(ns)\n";
    for(int i = 0; i < repetitions; i++){
        ofs << "Selection Sort," << i+1 << ",";
        shuffleArrays();       // load arrays with unsorted data
        selectionSorting(ofs); // sort all 6 arrays and write a line of results to sortResults.csv
    }
    for(int i = 0; i < repetitions; i++){
        ofs << "Heap Sort," << i+1 << ",";
        shuffleArrays();       // load arrays with unsorted data
        heapSorting(ofs);      // sort all 6 arrays and write a line of results to sortResults.csv
    }
    for(int i = 0; i < repetitions; i++){
        ofs << "Quick Sort," << i+1 << ",";
        shuffleArrays();       // load arrays with unsorted data
        quickSorting(ofs);     // sort all 6 arrays and write a line of results to sortResult.csv
    }
    ofs.close();
}
void sortAlgos::selectionSorting(ofstream& ofs){
    // sort List10.txt with selection sort
    start = chrono::high_resolution_clock::now();
    selectionSort(sort10, 10);
    stop = chrono::high_resolution_clock::now();
    time1micro = duration_cast<TIME_UNIT2>(stop - start);
    ofs << time1micro.count() << "," << count << ",";

    // sort List100.txt with selection sort
    start = chrono::high_resolution_clock::now();
    selectionSort(sort100, 100);
    stop = chrono::high_resolution_clock::now();
    time2micro = duration_cast<TIME_UNIT2>(stop - start);
    ofs << time2micro.count() << "," << count << ",";

    // sort List1000.txt with selection sort
```

```
63        start = chrono::high_resolution_clock::now();
64        selectionSort(sort1000, 1000);
65        stop = chrono::high_resolution_clock::now();
66        time3micro = duration_cast<TIME_UNIT2>(stop - start);
67        ofs << time3micro.count() << "," << count << ",";
68
69        // sort List10000.txt with selection sort
70        start = chrono::high_resolution_clock::now();
71        selectionSort(sort10000, 10000);
72        stop = chrono::high_resolution_clock::now();
73        time4micro = duration_cast<TIME_UNIT2>(stop - start);
74        ofs << time4micro.count() << "," << count << ",";
75
76        // sort List100000.txt with selection sort
77        start = chrono::high_resolution_clock::now();
78        selectionSort(sort100000, 100000);
79        stop = chrono::high_resolution_clock::now();
80        time5micro = duration_cast<TIME_UNIT2>(stop - start);
81        ofs << time5micro.count() << "," << count << ",";
82
83        // sort List250000.txt with selection sort
84        start = chrono::high_resolution_clock::now();
85        selectionSort(sort250000, 250000);
86        stop = chrono::high_resolution_clock::now();
87        time6micro = duration_cast<TIME_UNIT2>(stop - start);
88        ofs << time6micro.count() << "," << count << ",";
89
90        // average of the 6 sorts
91        ofs << 1.0*(time1micro.count() + time2micro.count() + time3micro.count() +
          time4micro.count() + time5micro.count() + time6micro.count())/6 << endl;
92    }
93    void sortAlgos::heapSorting(ofstream& ofs){
94        // sort List10.txt with heap sort
95        start = chrono::high_resolution_clock::now();
96        heapSort(sort10, 10);
97        stop = chrono::high_resolution_clock::now();
98        time1 = duration_cast<TIME_UNIT>(stop - start);
99        ofs << time1.count() << "," << count << ",";
100
101       // sort List100.txt with heap sort
102       start = chrono::high_resolution_clock::now();
103       heapSort(sort100, 100);
104       stop = chrono::high_resolution_clock::now();
105       time2 = duration_cast<TIME_UNIT>(stop - start);
106       ofs << time2.count() << "," << count << ",";
107
108       // sort List1000.txt with heap sort
109       start = chrono::high_resolution_clock::now();
110       heapSort(sort1000, 1000);
111       stop = chrono::high_resolution_clock::now();
112       time3 = duration_cast<TIME_UNIT>(stop - start);
113       ofs << time3.count() << "," << count << ",";
114
115       // sort List10000.txt with heap sort
116       start = chrono::high_resolution_clock::now();
117       heapSort(sort10000, 10000);
118       stop = chrono::high_resolution_clock::now();
119       time4 = duration_cast<TIME_UNIT>(stop - start);
120       ofs << time4.count() << "," << count << ",";
121
122       // sort List100000.txt with heap sort
123       start = chrono::high_resolution_clock::now();
124       heapSort(sort100000, 100000);
125       stop = chrono::high_resolution_clock::now();
126       time5 = duration_cast<TIME_UNIT>(stop - start);
127       ofs << time5.count() << "," << count << ",";
128
129       // sort List250000.txt with heap sort
130       start = chrono::high_resolution_clock::now();
```

```
131         heapSort(sort250000, 250000);
132         stop = chrono::high_resolution_clock::now();
133         time6 = duration_cast<TIME_UNIT>(stop - start);
134         ofs << time6.count() << "," << count << ",";
135 
136         // average of the 6 sorts
137         ofs << 1.0*(time1.count() + time2.count() + time3.count() + time4.count() +
            time5.count() + time6.count())/6 << endl;
138     }
139     void sortAlgos::quickSorting(ofstream& ofs){
140         // sort List10.txt with quick sort
141         start = chrono::high_resolution_clock::now();
142         count = 0;
143         count = quickSort(sort10, 0, 9);
144         stop = chrono::high_resolution_clock::now();
145         time1 = duration_cast<TIME_UNIT>(stop - start);
146         ofs << time1.count() << "," << count << ",";
147 
148         // sort List100.txt with quick sort
149         start = chrono::high_resolution_clock::now();
150         count = 0;
151         count = quickSort(sort100, 0, 99);
152         stop = chrono::high_resolution_clock::now();
153         time2 = duration_cast<TIME_UNIT>(stop - start);
154         ofs << time2.count() << "," << count << ",";
155 
156         // sort List1000.txt with quick sort
157         start = chrono::high_resolution_clock::now();
158         count = 0;
159         count = quickSort(sort1000, 0, 999);
160         stop = chrono::high_resolution_clock::now();
161         time3 = duration_cast<TIME_UNIT>(stop - start);
162         ofs << time3.count() << "," << count << ",";
163 
164         // sort List10000.txt with quick sort
165         start = chrono::high_resolution_clock::now();
166         count = 0;
167         count = quickSort(sort10000, 0, 9999);
168         stop = chrono::high_resolution_clock::now();
169         time4 = duration_cast<TIME_UNIT>(stop - start);
170         ofs << time4.count() << "," << count << ",";
171 
172         // sort List100000.txt with quick sort
173         start = chrono::high_resolution_clock::now();
174         count = 0;
175         count = quickSort(sort100000, 0, 99999);
176         stop = chrono::high_resolution_clock::now();
177         time5 = duration_cast<TIME_UNIT>(stop - start);
178         ofs << time5.count() << "," << count << ",";
179 
180         // sort List250000.txt with quick sort
181         start = chrono::high_resolution_clock::now();
182         count = 0;
183         count = quickSort(sort250000, 0, 249999);
184         stop = chrono::high_resolution_clock::now();
185         time6 = duration_cast<TIME_UNIT>(stop - start);
186         ofs << time6.count() << "," << count << ",";
187 
188         // average of the 6 sorts
189         ofs << 1.0*(time1.count() + time2.count() + time3.count() + time4.count() +
            time5.count() + time6.count())/6 << endl;
190     }
191     int sortAlgos::selectionSort(int arr[], int arrSize){
192         int min, minPos;
193         count = 0;
194         for(int i = 0; i < (arrSize-1); i++){
195             minPos = i;
196             min = arr[i];
197             for(int j = i+1; j < arrSize; j++){
```

```
198             if(arr[j] < min){
199                 minPos = j;
200                 min = arr[j];
201             }
202             count++; // counts the number of iterations
203         }
204         arr[minPos] = arr[i];
205         arr[i] = min;
206     }
207     return count;
208 }
209 int sortAlgos::heapSort(int arr[], int arrSize){
210     // create a maxheap from the input array
211     count = 0;
212     for(int i = arrSize / 2 - 1; i >= 0; i--){
213         heapify(arr, arrSize, i);
214     }
215 
216     // extract the max (root) from the heap and move it to the end then rebuild the
    remaining heap
217     for(int i = arrSize - 1; i >= 0; i--){
218         // swap max to end
219         swap(arr[0], arr[i]);
220 
221         // call heapify to fix heap
222         heapify(arr, i, 0);
223     }
224     return count;
225 }
226 void sortAlgos::heapify(int arr[], int heapSize, int root){
227     // largest, root, left, and right are indexes of elements in the heap
228     int largest = root;         // initialize largest as root
229     int left = 2*root + 1;      // finds the left "node" of a given root
230     int right = 2*root + 2;     // finds the right "node" of a given root
231 
232     // checks if the left or right node is larger than the given root
233     if(left < heapSize && arr[left] > arr[largest]){
234         largest = left;
235     }
236     if(right < heapSize && arr[right] > arr[largest]){
237         largest = right;
238     }
239 
240     // if the one of the nodes was larger than the root, then swap it with the root
241     if(largest != root){
242         swap(arr[root], arr[largest]);
243         heapify(arr, heapSize, largest);
244     }
245     count++;     // counts the number of heapify calls
246 }
247 int sortAlgos::quickSort(int arr[], int low, int high){
248     if(low < high){
249         int partIndex = partition(arr, low, high);
250         quickSort(arr, low, partIndex - 1);
251         quickSort(arr, partIndex + 1, high);
252     }
253     return count;
254 }
255 int sortAlgos::partition(int arr[], int low, int high){
256     int pivot = arr[high];  // set pivot
257     int i = (low - 1);      // index of smaller element
258     count++;                // count the number of quickSort calls
259     for(int j = low; j <= high - 1; j++){
260         // if current element is smaller than of equal to pivot
261         if(arr[j] <= pivot){
262             i++;    // increment indox of smaller element
263             flip(&arr[i], &arr[j]);
264         }
265     }
```

```
266         flip(&arr[i + 1], &arr[high]);
267         return(i + 1);
268     }
269     void sortAlgos::flip(int* a, int* b) {
270         int t = *a;
271         *a = *b;
272         *b = t;
273     }
274     void sortAlgos::printArray(int arr[], int arrSize){
275         for(int i = 0; i < arrSize; i++){
276             cout << arr[i] << "\t";
277             if(i % 9 == 0) cout << endl;
278         }
279         cout << endl;
280     }
281
```