Linear Search

The linear search traverses the array one by one from the beginning. It works better for smaller data sets than larger ones. Here is the code for my linear search. The first argument for all of these functions is a pointer to an array.

```
/* Sorting and Searching: Linear Search
 * CIS 152 - Data Structures
 * Tanner Babcock
 * October 16, 2022 */
#include <iostream>
#include "searchingSorting.h"

/* This is linear search. It traverses the array one by one from the beginning.
 * It works better for smaller data sets than larger ones. */
int linearSearch(int *a, int size, int x) {
    /* iterate through elements */
    for (int y = 0; y < size; y++) {
        if (a[y] == x)
            return y;
    }
    return -1;
}

```

Binary Search

My binary search function is recursive. The binary search splits the array in half, down the middle. This only works if the values of the array are in order from smallest to largest. I believe that the binary search works better than the linear search, if it has appropriate data to work with. This is the code for my binary search.

```
/* Sorting and Searching Experiment: Binary Search
 * Tanner Babcock
 * CIS 152 - Data Structures
 * October 16, 2022 */
#include <iostream>
#include "sortingSearching.h"

/* This is binary search. It is recursive, and splits the array in half
 * down the middle. This only works if the values of the array are in
 * order from smallest to largest. */
int binarySearch(int *a, int low, int high, int x) {
    if (high >= low) {
        int mid = (low + (high - low) / 2);

        if (a[mid] == x)
            return mid;

        /* present in left half */
        if (a[mid] > x)
            return binarySearch(a, low, mid - 1, x);

        /* else present in right half */
        return binarySearch(a, mid + 1, high, x);
    }
    /* element is not present */
    return -1;
}
```

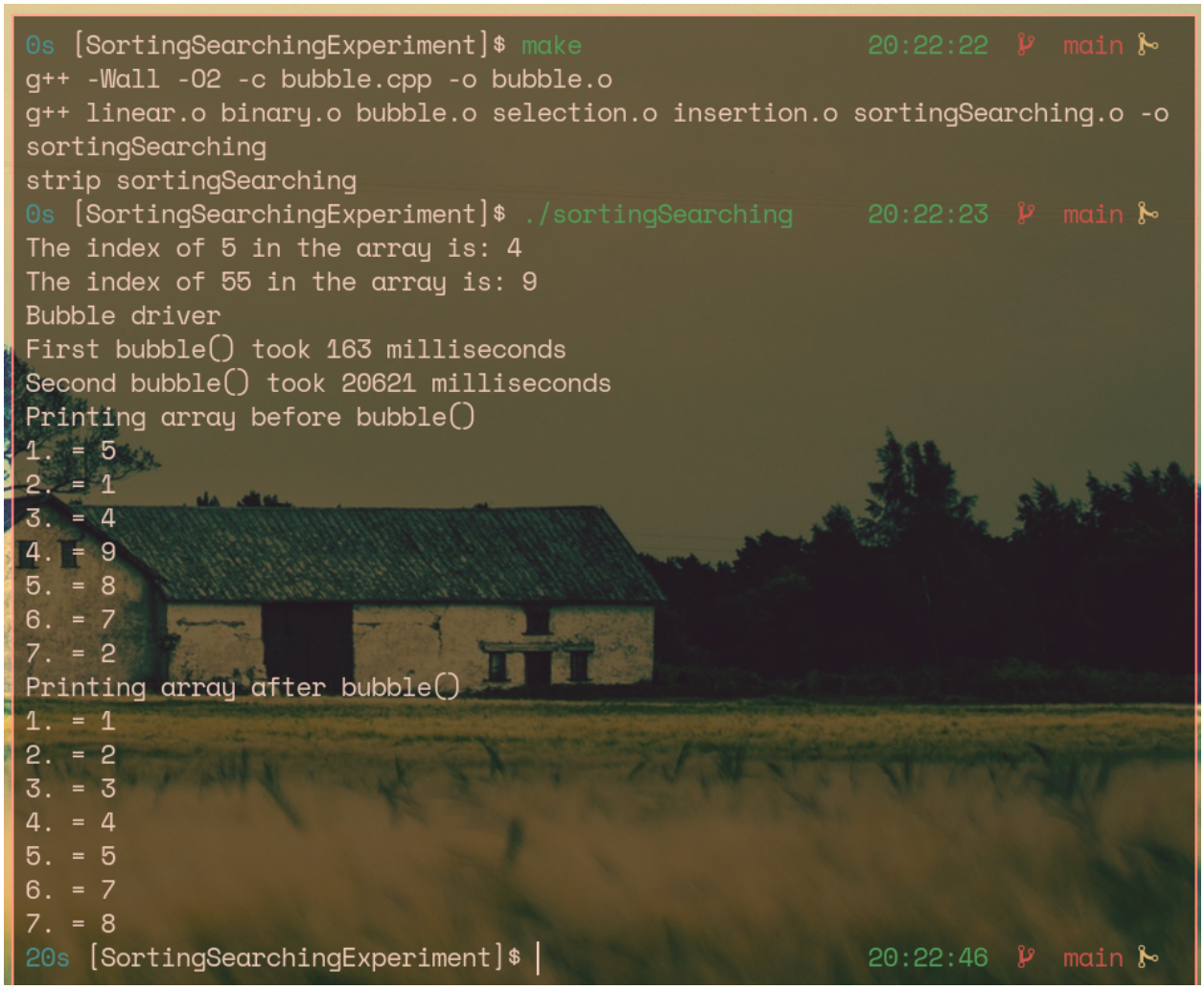
Bubble Sort

The bubble sort method compares the value of each element of the array with the value of the element after it. If the first value is greater than the second, it swaps the values, so the greater value comes later. The bubble sort is the slowest sorting algorithm of the three. Here is the code.

```
/* Sorting and Searching Experiment: Bubble Sort
 * CIS 152 - Data Structures
 * Tanner Babcock
 * October 16, 2022 */
#include <cstdlib>
#include <ctime>
#include "sortingSearching.h"

/* The bubble sort compares the value of each element
 * of the array with the value of the element after it.
 * If the first value is greater than the second, swap
 * the values, so the greater one comes later. */
void bubble(int *a, int size) {
    for (int x = 0; x < (size - 1); x++) {
        for (int y = 0; y < (size - x) - 1; y++) {
            if (a[y] > a[y+1])
                swap(&a[y], &a[y+1]);
        }
    }
}

void bubbleDriver(void) {
    int arr[10000];
    srand(time(NULL));

    for (int i = 0; i < 10000; i++) {
        arr[i] = rand() % 20;
    }
    printArray(arr, 10000);
    bubble(arr, 10000);
    printArray(arr, 10000);
}
```

This is not the finished bubble sort driver function, just a basic one to start out the testing. Here are the results of the first test.

Here are the results of the second test.



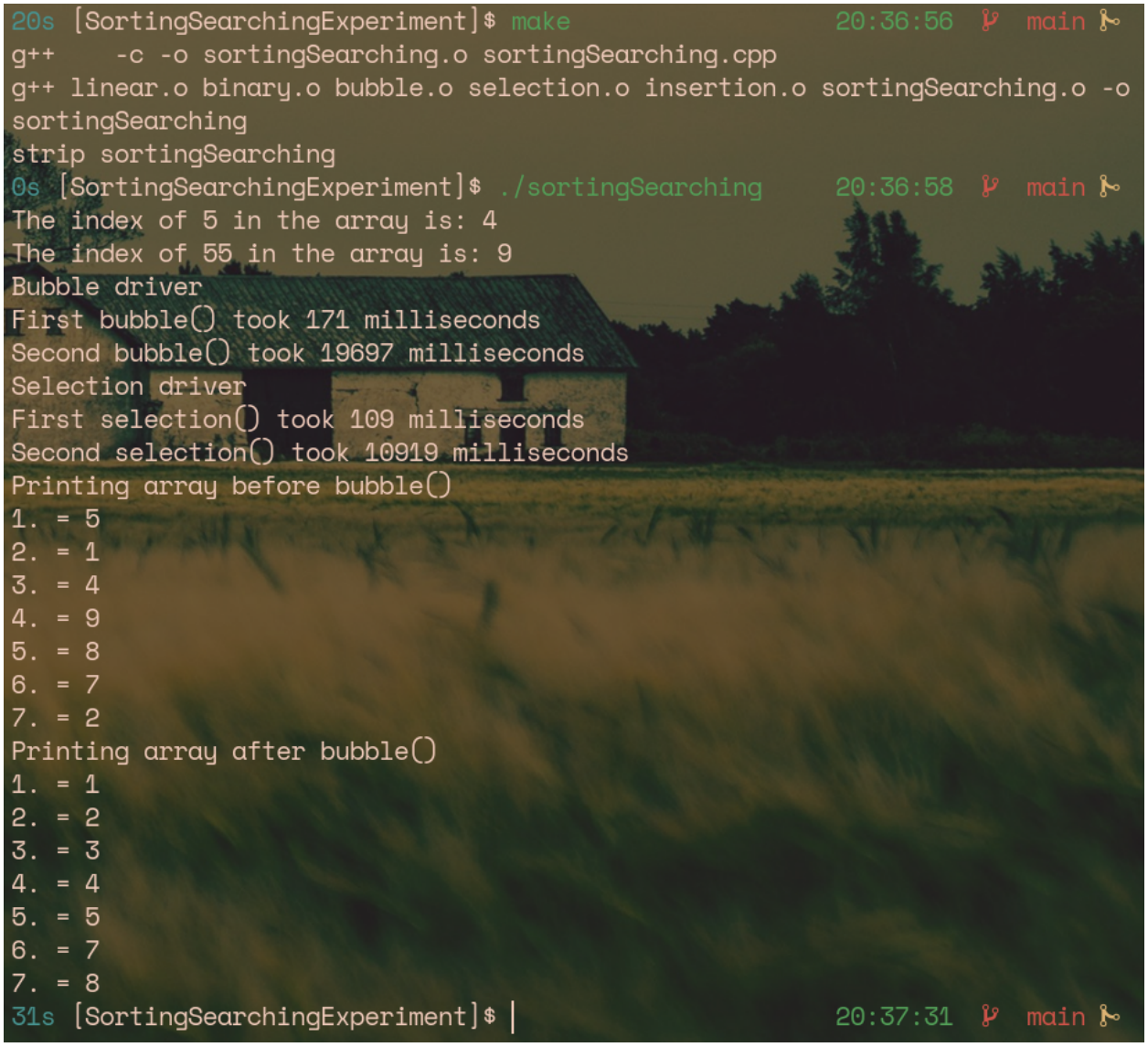
Bubble-sorting the first array takes less than a second, and sorting the second array can take up to 20 seconds.

Selection Sort

The selection sort compares the value of each element of an array with the value of each other element, to try to find the minimum and place it at the beginning. It is the second-fastest sorting algorithm after the insertion sort, and is good for large sets of data. Here is my selection sort function.

```
/* Sorting and Searching Experiment: Selection Sort
 * CIS 152 - Data Structures
 * Tanner Babcock
 * October 16, 2022 */
#include <chrono>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include "sortingSearching.h"

/* The selection sort compares the value of each
 * element with the value of each other element, to
 * try to find the minimum and place at the
 * beginning. */
void selection(int *a, int size) {
    int min;
    for (int x = 0; x < (size - 1); x++) {
        /* find the minimum element */
        min = x;
        for (int y = x+1; y < size; y++) {
            if (a[y] < a[min])
                min = y;
        }

        /* swap elements */
        if (min != x)
            swap(&a[min], &a[x]);
    }
}
```

Here are the results of the tests.



Here is the driver function for the selection sort.

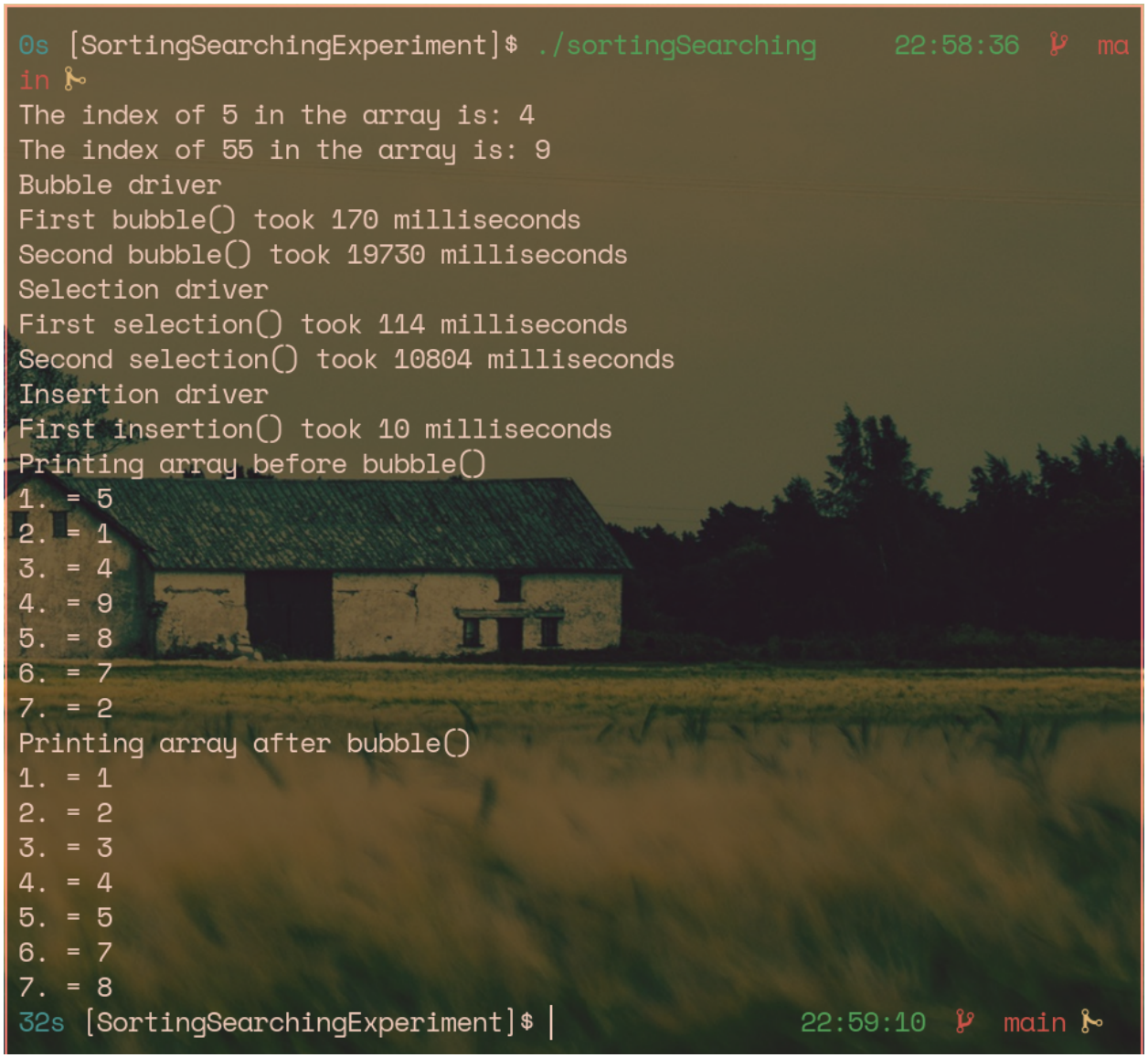
```
            swap(&a[min], &a[x]);
        }
    }

void selectionDriver(void) {
    auto start = std::chrono::high_resolution_clock::now();
    int arr[10000];
    srand(time(NULL));

    for (int i = 0; i < 10000; i++) {
        arr[i] = rand() % 30;
    }
    selection(arr, 10000);

    auto stop = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(stop -
    std::cout << "First selection() took " << duration.count() << " milliseconds'

    start = std::chrono::high_resolution_clock::now();
    int arr2[100000];
    srand(time(NULL));

    for (int i = 0; i < 100000; i++) {
        arr2[i] = rand() % 30;
    }
    selection(arr2, 100000);
    stop = std::chrono::high_resolution_clock::now();
    duration = std::chrono::duration_cast<std::chrono::milliseconds>(stop - start
    std::cout << "Second selection() took " << duration.count() << " milliseconds
}
```

Insertion Sort

The insertion sort is good for data sets that are already partially sorted. It is the simplest sorting algorithm, and the fastest, completing the tests much faster than the bubble and selection sorting algorithms.

```cpp
/* Sorting and Searching Experiment: Insertion Sort
 * CIS 152 - Data Structures
 * Tanner Babcock
 * October 16, 2022 */
#include <chrono>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include "sortingSearching.h"

/* The insertion sort is good for data sets that are
 * already sorted. It is the simplest sorting algorithm. */
void insertion(int *a, int size) {
    int key, y = 0;
    for (int x = 1; x < size; x++) {
        key = a[x];
        y = x - 1;

        while ((y >= 0) && (a[y] > key)) {
            a[y+1] = a[y];
            y--;
        }
        a[y+1] = key;
    }
}

void insertionDriver(void) {
    auto start = std::chrono::high_resolution_clock::now();
    int arr[TRIALONE]; /* 10000 */
```

Here are the first set of results for the insertion sort driver.



Here are the second set of results for the insertion sorting algorithm.

Conclusion

As you can see, the insertion sort is the fastest sorting algorithm, and the simplest. With the selection sort being the second-slowest, and the bubble sort being the slowest, the insertion sort may seem like the best algorithm for the job, but it requires the data set to be already partially sorted. The bubble sort is the most complex of the three sorting

algorithms. It was necessary for me to print these large arrays so I could see if the sorting algorithms were working. I populated all of the large testing arrays with random numbers between 0 and 30, so that it would be easy to tell when an array is sorted or not, just by glancing at it. I learned a lot doing this assignment and now I feel I completely understand these sorting and searching algorithms. I will attach a ZIP file of my project to this assignment.