

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.

```
1 /* Sorting and Searching: Linear Search
2  * CIS 152 - Data Structures
3  * Tanner Babcock
4  * October 16, 2022 */
5 #include <iostream>
6 #include "searchingSorting.h"
7
8 /* This is linear search. It traverses the array one by one from the beginning.
9  * It works better for smaller data sets than larger ones. */
10 int linearSearch(int *a, int size, int x) {
11     /* iterate through elements */
12     for (int y = 0; y < size; y++) {
13         if (a[y] == x)
14             return y;
15     }
16     return -1;
17 }
```

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.

Babcock 2
CIS 152 - Data Structures

```
10 /* Sorting and Searching Experiment: Binary Search
 9  * Tanner Babcock
 8  * CIS 152 - Data Structures
 7  * October 16, 2022 */
 6 #include <iostream>
 5 #include "sortingSearching.h"
 4
 3 /* This is binary search. It is recursive, and splits the array in half
 2 * down the middle. This only works if the values of the array are in
 1 * order from smallest to largest. */
11 int binarySearch(int *a, int low, int high, int x) {
 1     if (high >= low) {
 2         int mid = (low + (high - low)) / 2;
 3
 4         if (a[mid] == x)
 5             return mid;
 6
 7         /* present in left half */
 8         if (a[mid] > x)
 9             return binarySearch(a, low, mid - 1, x);
10
11         /* else present in right half */
12         return binarySearch(a, mid + 1, high, x);
13     }
14     /* element is not present */
15     return -1;
16 }
```

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.

Babcock 3

CIS 152 - Data Structures

```
12 /* Sorting and Searching Experiment: Bubble Sort
11  * CIS 152 - Data Structures
10  * Tanner Babcock
9   * October 16, 2022 */
8 #include <cstdlib>
7 #include <ctime>
6 #include "sortingSearching.h"
5
4 /* The bubble sort compares the value of each element
3  * of the array with the value of the element after it.
2  * If the first value is greater than the second, swap
1  * the values, so the greater one comes later. */
13 void bubble(int *a, int size) {
1    for (int x = 0; x < (size - 1); x++) {
2        for (int y = 0; y < (size - x) - 1; y++) {
3            if (a[y] > a[y+1])
4                swap(&a[y], &a[y+1]);
5        }
6    }
7 }
8
9 void bubbleDriver(void) {
10    int arr[10000];
11    srand(time(NULL));
12
13    for (int i = 0; i < 10000; i++) {
14        arr[i] = rand() % 20;
15    }
16    printArray(arr, 10000);
17    bubble(arr, 10000);
18    printArray(arr, 10000);
19 }
```

Babcock 4

CIS 152 - Data Structures

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.

```
0s [SortingSearchingExperiment]$ ./sortingSearching      20:19:23 ✘ main ✘
The index of 5 in the array is: 4
The index of 55 in the array is: 9
Bubble driver
First bubble() took 175 milliseconds
Printing array before bubble()
1. = 5
2. = 1
3. = 4
4. = 9
5. = 8
6. = 7
7. = 2
Printing array after bubble()
1. = 1
2. = 2
3. = 3
4. = 4
5. = 5
6. = 7
7. = 8
0s [SortingSearchingExperiment]$ | 20:19:24 ✘ main ✘
```

Here are the results of the second test.

```
0s [SortingSearchingExperiment]$ make          20:22:22 ⌂ main ⌈
g++ -Wall -O2 -c bubble.cpp -o bubble.o
g++ linear.o binary.o bubble.o selection.o insertion.o sortingSearching.o -o
sortingSearching
strip sortingSearching
0s [SortingSearchingExperiment]$ ./sortingSearching      20:22:23 ⌂ main ⌈
The index of 5 in the array is: 4
The index of 55 in the array is: 9
Bubble driver
First bubble() took 163 milliseconds
Second bubble() took 20621 milliseconds
Printing array before bubble()
1. = 5
2. = 1
3. = 4
4. = 9
5. = 8
6. = 7
7. = 2
Printing array after bubble()
1. = 1
2. = 2
3. = 3
4. = 4
5. = 5
6. = 7
7. = 8
20s [SortingSearchingExperiment]$ |          20:22:46 ⌂ main ⌈
```

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.

Babcock 6

CIS 152 - Data Structures

```
14 /* Sorting and Searching Experiment: Selection Sort
13 * CIS 152 - Data Structures
12 * Tanner Babcock
11 * October 16, 2022 */
10 #include <chrono>
 9 #include <cstdlib>
 8 #include <ctime>
 7 #include <iostream>
 6 #include "sortingSearching.h"
 5
 4 /* The selection sort compares the value of each
 3 * element with the value of each other element, to
 2 * try to find the minimum and place at the
 1 * beginning. */
15 void selection(int *a, int size) {
 1     int min;
 2     for (int x = 0; x < (size - 1); x++) {
 3         /* find the minimum element */
 4         min = x;
 5         for (int y = x+1; y < size; y++) {
 6             if (a[y] < a[min])
 7                 min = y;
 8         }
 9
10         /* swap elements */
11         if (min != x)
12             swap(&a[min], &a[x]);
13     }
14 }
```

Here are the results of the tests.

```
20s [SortingSearchingExperiment]$ make          20:36:56 ✘ main ↵
g++ -c -o sortingSearching.o sortingSearching.cpp
g++ linear.o binary.o bubble.o selection.o insertion.o sortingSearching.o -o
sortingSearching
strip sortingSearching
0s [SortingSearchingExperiment]$ ./sortingSearching      20:36:58 ✘ main ↵
The index of 5 in the array is: 4
The index of 55 in the array is: 9
Bubble driver
First bubble() took 171 milliseconds
Second bubble() took 19697 milliseconds
Selection driver
First selection() took 109 milliseconds
Second selection() took 10919 milliseconds
Printing array before bubble()
1. = 5
2. = 1
3. = 4
4. = 9
5. = 8
6. = 7
7. = 2
Printing array after bubble()
1. = 1
2. = 2
3. = 3
4. = 4
5. = 5
6. = 7
7. = 8
31s [SortingSearchingExperiment]$ |          20:37:31 ✘ main ↵
```

Here is the driver function for the selection sort.

Babcock 8
CIS 152 - Data Structures

```
4             swap(&a[min], &a[x]);
3     }
2 }
1
31 void selectionDriver(void) {
1     auto start = std::chrono::high_resolution_clock::now();
2     int arr[10000];
3     srand(time(NULL));
4
5     for (int i = 0; i < 10000; i++) {
6         arr[i] = rand() % 30;
7     }
8     selection(arr, 10000);
9
10    auto stop = std::chrono::high_resolution_clock::now();
11    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(stop -
12    std::cout << "First selection() took " << duration.count() << " milliseconds"
13
14    start = std::chrono::high_resolution_clock::now();
15    int arr2[100000];
16    srand(time(NULL));
17
18    for (int i = 0; i < 100000; i++) {
19        arr2[i] = rand() % 30;
20    }
21    selection(arr2, 100000);
22    stop = std::chrono::high_resolution_clock::now();
23    duration = std::chrono::duration_cast<std::chrono::milliseconds>(stop - start);
24    std::cout << "Second selection() took " << duration.count() << " milliseconds"
25 }
```

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.

Babcock 9
CIS 152 - Data Structures

```
1 /* Sorting and Searching Experiment: Insertion Sort
2  * CIS 152 - Data Structures
3  * Tanner Babcock
4  * October 16, 2022 */
5 #include <chrono>
6 #include <cstdlib>
7 #include <ctime>
8 #include <iostream>
9 #include "sortingSearching.h"
10
11 /* The insertion sort is good for data sets that are
12  * already sorted. It is the simplest sorting algorithm. */
13 void insertion(int *a, int size) {
14     int key, y = 0;
15     for (int x = 1; x < size; x++) {
16         key = a[x];
17         y = x - 1;
18         while ((y >= 0) && (a[y] > key)) {
19             a[y+1] = a[y];
20             y--;
21         }
22         a[y+1] = key;
23     }
24 }
25
26 void insertionDriver(void) {
27     auto start = std::chrono::high_resolution_clock::now();
28     int arr[TRIALONE]; /* 10000 */
```

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

Babcock 10
CIS 152 - Data Structures

```
0s [SortingSearchingExperiment]$ ./sortingSearching      22:58:36 ⚡ main
in ↵
The index of 5 in the array is: 4
The index of 55 in the array is: 9
Bubble driver
First bubble() took 170 milliseconds
Second bubble() took 19730 milliseconds
Selection driver
First selection() took 114 milliseconds
Second selection() took 10804 milliseconds
Insertion driver
First insertion() took 10 milliseconds
Printing array before bubble()
1. = 5
2. = 1
3. = 4
4. = 9
5. = 8
6. = 7
7. = 2
Printing array after bubble()
1. = 1
2. = 2
3. = 3
4. = 4
5. = 5
6. = 7
7. = 8
32s [SortingSearchingExperiment]$ | 22:59:10 ⚡ main ↵
```

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

```
g++ -Wall -O2 -c insertion.cpp -o insertion.o
g++      -c -o sortingSearching.o sortingSearching.cpp
g++ linear.o binary.o bubble.o selection.o insertion.o sortingSearching
.o -o sortingSearching
strip sortingSearching
2s [SortingSearchingExperiment]$ ./sortingSearching
The index of 5 in the array is: 4
The index of 55 in the array is: 9
Bubble driver
First bubble() took 166 milliseconds
Second bubble() took 20658 milliseconds
Selection driver
First selection() took 117 milliseconds
Second selection() took 11710 milliseconds
Insertion driver
First insertion() took 10 milliseconds
Second insertion() took 1042 milliseconds
Printing array before bubble()
1. = 5
2. = 1
3. = 4
4. = 9
5. = 8
6. = 7
7. = 2
Printing array after bubble()
1. = 1
2. = 2
3. = 3
4. = 4
5. = 5
6. = 7
7. = 8
34s [SortingSearchingExperiment]$ | 23:04:00 ↵ main ↵
```

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

Babcock 12

CIS 152 - Data Structures

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.