Physics 2G03 - Homework 7

1.1

Please see the source file: *Makefile*

1.2

Please see the following source files:
Makefile, *sort.h*, *sort.cpp*, *testsort.cpp*

1.3

Please see the source files: *sort.cpp* & *sort.h*

1.4

```
// Swap A[i] and A[j];
A[i] = A[j]; // Step 1
A[j] = A[i]; // Step 2
```

The above code is incorrect because the value of A[i] is lost/replaced before it is assigned to A[j]. In Step 1, the value of A[j] is assigned to A[i]. Now A[j] and A[i] hold the same value. In Step 2, the value of A[i] is assigned to A[j]. However, this is incorrect because A[j] and A[j] hold the same value, and the original value for A[i] is now lost. The following block of code demonstrates this:

```
A[i] = 2; // Sets the value of A[i] to 2
A[j] = 3; // Sets the value of A[j] to 3
// Both A[i] and A[j] are unique and hold different values

A[i] = A[j]; // Step 1
/*
  In Step 1, the value of A[j] is assigned to A[i]
  Thus, A[i] now holds 3
  Since the original value of A[i] was not saved anywhere,
    it is now lost
  A[i] and A[j] hold the same value; a value of 3
*/
A[j] = A[i]; // Step 2
/*
  In Step 2, the value of A[i] is assigned to A[j]
  Since A[j] and A[j] hold the same value, A[j] does not change
*/
```

To fix this issue, you need to create a temporary variable to store the value of A[i] before any value is assigned to it. Then you can reference the temporary variable and assign it to A[j]. For example:

```
int tempVar = A[i]; // Step 1
A[i] = A[j];        // Step 2
A[j] = tempVar;     // Step 3
```

Note: The temporary variable must be the same data type as the array

1.5

Please see the source files: *sort.cpp* & *sort.h*

1.6

Please see the source files: *sort.cpp* & *sort.h*

1.7

Trials for InsertionSort:

```
1) 0.149u 0.002s 0:04.68 2.9%
2) 0.135u 0.001s 0:02.31 5.6%
3) 0.133u 0.003s 0:01.13 11.5%
4) 0.125u 0.001s 0:01.06 11.3%
5) 0.134u 0.001s 0:01.08 12.0%
```

Trials For QuickSort:

```
1) 0.004u 0.001s 0:01.14 0.0%
2) 0.004u 0.002s 0:01.09 0.0%
3) 0.003u 0.001s 0:01.09 0.0%
4) 0.004u 0.000s 0:01.15 0.0%
5) 0.005u 0.001s 0:01.28 0.0%
```

It appears that both QuickSort and InsertionSort are very quick and use very little CPU-seconds compared to each other. However, after taking a look at the other numbers, it seems that QuickSort is true to its name and performs quicker than InsertionSort.

1.8

When the array is ordered top-down:

InsertionSort performs better than QuickSort because the array is already sorted (or mostly sorted) and InsertionSort needs to iterate through the entire array once (or a few more times). On the other hand, QuickSort breaks up the array and begins sorting it. This makes it slightly more slower than InsertionSort, but still quick.

When the array is ordered bottom-up:

InsertionSort performs (way) worse than QuickSort because it has to do many many more iterations than QuickSort. The increased number of iterations slows down the algorithm and makes it slower than QuickSort. This is because it needs to take each value from the bottom and move it to the top and iterate through every subsequent value.