Eric Watson

ECE 331 Homework 3

This assignment required us to implement the insertion sort and merge sort algorithms as c functions, and then compare the execution times upon testing the algorithms with integer arrays of size 10, 100, and 1000. To ensure a valid comparison, the same array of randomly generated integers was passed to both functions. For both the insertion sort and merge sort functions I followed the pseudo code provided in the textbook very closely. Within my main function I populate an array with random integers, make a copy of the array for the second sorting algorithm, initiate and stop the execution timers when each sorting function is called, and then print the resulting sorted arrays to ensure each algorithm was sorting properly. In order to populate the array with random integers I utilized the c library functions *rand()* and *srand()*. Since our maximum test array size was only 1000, I made this the seed value so that the randomly generated numbers would be ranging from 0 to 1000. In order to obtain the execution times I used the C library timing function *clock()*.

The running times of the insertion sort and merge sort algorithms are $T(n)=\Theta(n^2)$ (worst case) and $T(n)=\Theta(n\lg(n))$ (worst case) respectively. Since the logarithm function grows more slowly than any linear function, for large enough inputs the merge sort outperforms insertion sort. Based on the approximated running times of these algorithms, as well as the results of my experimental trials, I would choose the insertion sort algorithm for smaller sized arrays, or arrays that are nearly sorted to begin with. For larger arrays, I would choose the merge sort algorithm. The results of the trials are shown in the table below.

Sorting Algorithm Execution Times

Array Size:	**10**	**100**	**1000**
Insertion Sort	4 μ seconds	25 μ seconds	1319 μ seconds
Merge Sort	15 μ seconds	41 μ seconds	265 μ seconds

The first terminal output on the following page demonstrates the full functionality of the program. For debugging purposes I print the original randomly generated array (which was copied for use with both algorithms), the execution times for each sorting algorithm, and also the resulting sorted arrays for both methods.

Testing Array Size of 10

```
Erics-MacBook-Pro-2:HW03 Watson$ gcc sort_compare.c
Erics-MacBook-Pro-2:HW03 Watson$ ./a.out
Input Array Size: 10
The randomly generated values are:
538
993
608
80
399
561
140
369
34
997
Insertion Sort Execution Time: 0.000004 seconds
The insertion sorted values are:
34
80
140
369
399
538
561
608
993
997
Merge Sort Execution Time: 0.000015 seconds
The merge sorted values are:
34
80
140
369
399
538
561
608
993
997
```

The ".c" file submitted with this document will have the array content debugging statements print only the first 10 array elements. This demonstrates that the algorithms are working properly, without creating a lengthy terminal output for large sized array. The output from testing arrays of size 100 and 1000 are shown below.

Testing Array Size of 100

```
Erics-MacBook-Pro-2:HW03 Watson$ ./a.out
Input Array Size: 100
Insertion Sort Execution Time: 0.000025 seconds
The first 10 insertion-sorted values are:
16
43
74
103
104
120
127
```

```
138
146
161
Merge Sort Execution Time: 0.000041 seconds
The first 10 merge-sorted values are:
16
43
74
103
104
120
127
138
146
161
```

Testing Array Size of 1000

```
Erics-MacBook-Pro-2:HW03 Watson$ gcc sort_compare.c
Erics-MacBook-Pro-2:HW03 Watson$ ./a.out
Input Array Size: 1000
Insertion Sort Execution Time: 0.001319 seconds
The first 10 insertion-sorted values are:
0
3
5
6
7
8
9
10
13
14
Merge Sort Execution Time: 0.000265 seconds
The first 10 merge-sorted values are:
0
3
5
6
7
8
9
10
13
14
```