Asymptotic Analysis Project

COT 4400 Spring 2018

My Nguyen, U71337744

1. OVERVIEW

The objective of this project is to evaluate and analyze the four following four algorithms: Selection Sort, Insertion Sort, Merge Sort, and Quick Sort. The purpose is to relate their theoretical runtime to their empirical (actual) behavior.

2. RESULTS

Each sorting algorithm has three associated types: constant, sorted, and random. These cases have arrays that are powers of 10 and there are 12 types for this project. For each type, the following metrics are recorded:

1. n_{min} =: the smallest power of 10 array size that takes 20 milliseconds or more per run to sort;
2. t_{min}: the time to sort an array of size n_{min};
3. n_{max}: the largest power of 10 array size that takes 10 minutes or less per run to sort (30 mins for all 3 runs), or the largest input size you can successfully sort if no input took more than 30 minutes total;
4. t_{max}: the time required to sort n_{max} elements.

From these metrics, the following results are generated as seen in Table 1.

3.1 Results				
type	**n_min**	**t_min (ms)**	**n_max**	**t_max (ms)**
SC	3800	20	600000	461014
SS	3900	20	600000	489301
SR	3790	20	600000	466977
IC	3900000	20	500000000	2615
IS	3900000	20	500000000	2402
IR	4200	20	800000	570579
MC	210000	20	1000000000	169252
MS	209000	20	1000000000	290515
MR	100000	20	295000000	89745
QC	5000	20	900000	581361
QS	280000	20	1000000000	141896
QR	88500	20	500000000	183126

Table 1: Sorting Algorithm Results

3. ANALYSIS

From the results of each algorithm shown in Table 2, an analysis of the estimate of the time complexity of each algorithm and its associated types will be made by comparing the ratio between t_{min} and t_{max} to the different well-known complexity of each algorithm. The three functions used to do the analysis are $f_1(n) = n,\ f_2(n) = n * \ln(n)\,, and\ f_3(n) = n^2$ which are linear, logarithmic, and quadratic, respectively.

3.2 Analysis					
type	**t_max/t_min**	**n ratio**	**n*ln(n) ratio**	**n^2 ratio**	**behavior**
SC	23051	158	255	24931	n^2
SS	24465	154	248	23669	n^2
SR	23349	158	256	25062	n^2
IC	131	128	169	16437	n
IS	120	128	169	16437	n
IR	28529	190	310	36281	n^2
MC	8463	4762	8052	22675737	n*ln(n)
MS	14526	4785	8094	22893249	n*ln(n)
MR	4487	2950	4997	8702500	n*ln(n)
QC	29068	180	290	32400	n^2
QS	7095	3571	5901	12755102	n*ln(n)
QR	9156	5650	9935	31919308	n*ln(n)

Table 2: Empirical Sorting Algorithm Analysis

3.1. SUMMARY

The behavior for each algorithm is as expected and similar to the analysis done in class with brute force and Master Theorem. The following summaries of each algorithm will use information shown in Table 2 and Table 3.

	Best-case complexity	Average-case complexity	Worst-case complexity
SelectionSort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
InsertionSort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
MergeSort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
QuickSort	$\Omega(n \lg n)$	$\Theta(n \lg n)$	$O(n^2)$

Table 3: Theoretical Sorting Algorithm Analysis

3.1.1 Selection Sort

Comparing the empirical results found in Table 2 and the theoretical results found in Table 3, the behavior of the Selection Sort for all three types were as expected with the time complexity of $\Theta(n^2)$. The following metrics were found for each of the types:

- constant type: $n_{min} = 3.80 * 10^3$, $t_{min} = 20\ ms$, $n_{max} = 6.00 * 10^5$, and $t_{max} = 4.61 * 10^5\ ms$, where the t_{max}/t_{min} ratio was closest to $\Theta(n^2)$;
- sorted type: $n_{min} = 3.90 * 10^3$, $t_{min} = 20\ ms$, $n_{max} = 6.00 * 10^5$, and $t_{max} = 4.89 * 10^5\ ms$, where the t_{max}/t_{min} ratio was closest to $\Theta(n^2)$;
- random type: $n_{min} = 3.79 * 10^3$, $t_{min} = 20\ ms$, $n_{max} = 6.00 * 10^5$, and $t_{max} = 4.67 * 10^5\ ms$, where the t_{max}/t_{min} ratio was closest to $\Theta(n^2)$.

The Selection Sort algorithm's empirical and theoretical behavior have all three time complexities: best-case, average-case, and worst-case. The empirical results match with the theoretical results because this algorithm iterates through the array linearly for *n* times with *n* inputs.

3.1.2 Insertion Sort

Comparing the empirical results found in Table 2 and the theoretical results found in Table 3, the behavior of the Insertion Sort for all three types were as expected. The following metrics were found for each of the types:

- constant type: $n_{min} = 3.90 * 10^6$, $t_{min} = 20\ ms$, $n_{max} = 5.00 * 10^8$, and $t_{max} = 2.62 * 10^3\ ms$, where the t_{max}/t_{min} ratio was closest to $\Theta(n)$;
- sorted type: $n_{min} = 3.90 * 10^6$, $t_{min} = 20\ ms$, $n_{max} = 5.00 * 10^8$, and $t_{max} = 2.40 * 10^3\ ms$, where the t_{max}/t_{min} ratio was closest to $\Theta(n)$;
- random type: $n_{min} = 4.20 * 10^3$, $t_{min} = 20\ ms$, $n_{max} = 8.00 * 10^5$, and $t_{max} = 5.71 * 10^5\ ms$, where the t_{max}/t_{min} ratio was closest to $\Theta(n^2)$.

The Insertion Sort constant and sorted types have the worst-case time complexity of $O(n)$ and the random type has the worst-case time complexity $O(n^2)$. The empirical results match with the theoretical results for constant and sorted because this algorithm iterates through the array by taking one input element, finding its location, and then inserting it into that spot. Thus for constant and sorted arrays, Insertion Sort has the worst-case time complexity because it checks a sorted list and inserts the value into its position. As for a random array, this algorithm has the worst-case time complexity because it has to sort through each element in the array and then insert the correct element in the correct position.

3.1.3 Merge Sort

Comparing the empirical results found in Table 2 and the theoretical results found in Table 3, the behavior of the Merge Sort for all three types were as expected. The following metrics were found for each of the types:

- constant type: $n_{min} = 2.10 * 10^5$, $t_{min} = 20\ ms$, $n_{max} = 1.00 * 10^9$, and $t_{max} = 1.69 * 10^5\ ms$, where the t_{max}/t_{min} ratio was closest to $\Theta(n * \ln(n))$;
- sorted type: $n_{min} = 2.09 * 10^5$, $t_{min} = 20\ ms$, $n_{max} = 1.00 * 10^9$, and $t_{max} = 2.91 * 10^5\ ms$, where the t_{max}/t_{min} ratio was closest to $\Theta(n * \ln(n))$;
- random type: $n_{min} = 1.00 * 10^5$, $t_{min} = 20\ ms$, $n_{max} = 2.95 * 10^8$, and $t_{max} = 8.97 * 10^4\ ms$, where the t_{max}/t_{min} ratio was closest to $\Theta(n * \ln(n))$.

Merge Sort algorithm's empirical and theoretical behavior have all three time complexities: best-case, average-case, and worst-case. For all three types associated with Merge Sort, the behavior converges to $\Theta(n * \ln(n))$ because the algorithm doesn't account for the type i.e. random, sorted, or constant. The Merge Sort splits an array into two halves and reclusively calls the algorithm of each half. Surprisingly, based on running the algorithm to gather data, there were some errors like 'segmentation fault' or 'incorrectly sorted array' due to not enough stack size or large number of inputs for *n*, size of the array.

3.1.4 Quick Sort

Comparing the empirical results found in Table 2 and the theoretical results found in Table 3, the behavior of the Quick Sort for all constant and random types were as expected. The following metrics were found for each of the types:

- constant type: $n_{min} = 5.00 * 10^3$, $t_{min} = 20\ ms$, $n_{max} = 9.00 * 10^5$, and $t_{max} = 5.81 * 10^5\ ms$, where the t_{max}/t_{min} ratio was closest to $\Theta(n^2)$;
- sorted type: $n_{min} = 2.80 * 10^5$, $t_{min} = 20\ ms$, $n_{max} = 1.00 * 10^9$, and $t_{max} = 1.41 * 10^5\ ms$, where the t_{max}/t_{min} ratio was closest to $\Theta(n * \ln(n))$
- random type: $n_{min} = 8.85 * 10^4$, $t_{min} = 20\ ms$, $n_{max} = 5.00 * 10^8$, and $t_{max} = 1.83 * 10^5\ ms$, where the t_{max}/t_{min} ratio was closest to $\Theta(n * \ln(n))$.

For the Quick Sort constant, the behavior converges to worst-case time complexity $\Theta(n^2)$ for constant type and average-case time complexity $\Theta(n * \ln(n))$ for sorted and random types. The Quick Sort algorithm choses a pivot value and splits the array at that point into two halves: smaller values on the left of the pivot and larger values on the right of the pivot. The constant type has the worst-case time complexity because this algorithm assumes that values are constant and unsorted within the array and it has to sort through the array and find the pivot value. As for the sorted and random types, the time complexity was average-case because the algorithm has to choose a pivot value close to the middle of the array and sort the array.

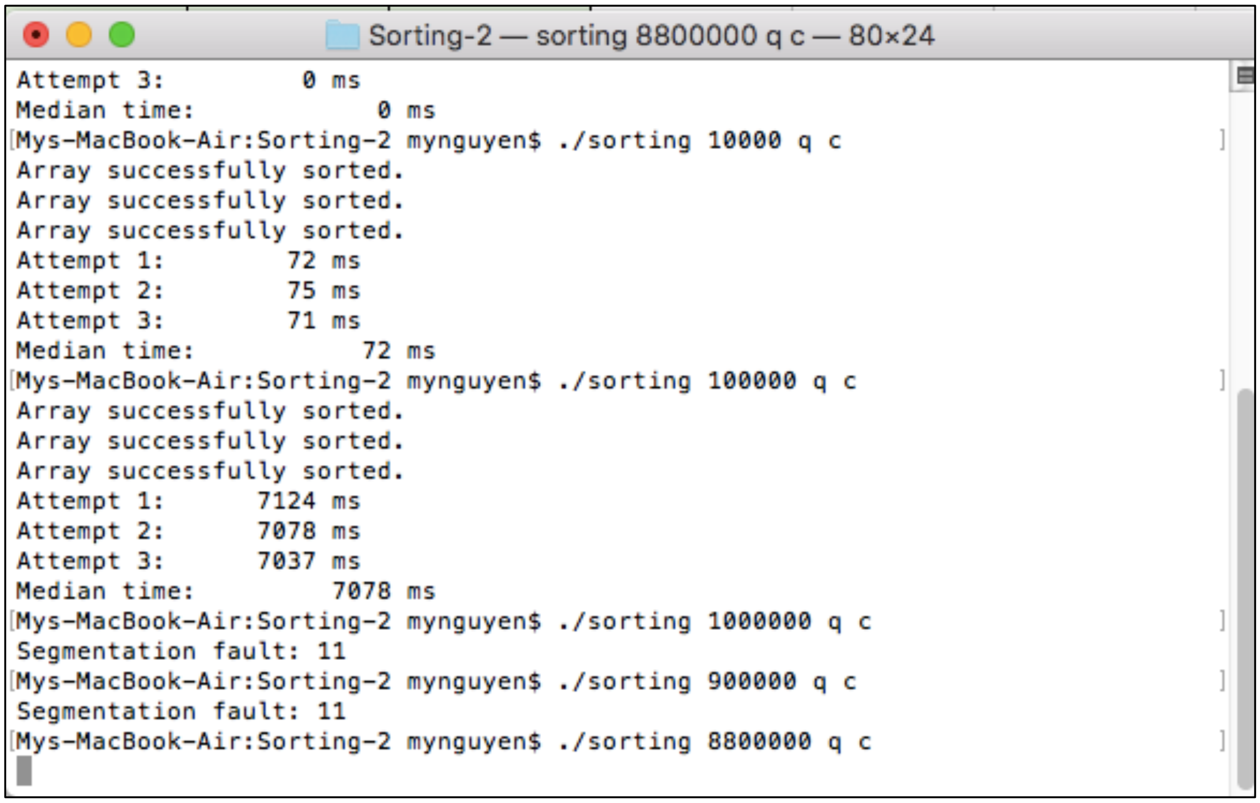
Surprisingly, based on running the algorithm to gather data, there were some errors like 'segmentation fault' or 'incorrectly sorted array' due to not enough stack size or large number of inputs for *n*, size of the array for constant and sorted types, which is shown below in Figures 1 and 2, respectively.

4. CONCLUSION

In conclusion, the project is easy to comprehend however, there were some issues that occurred during the process.

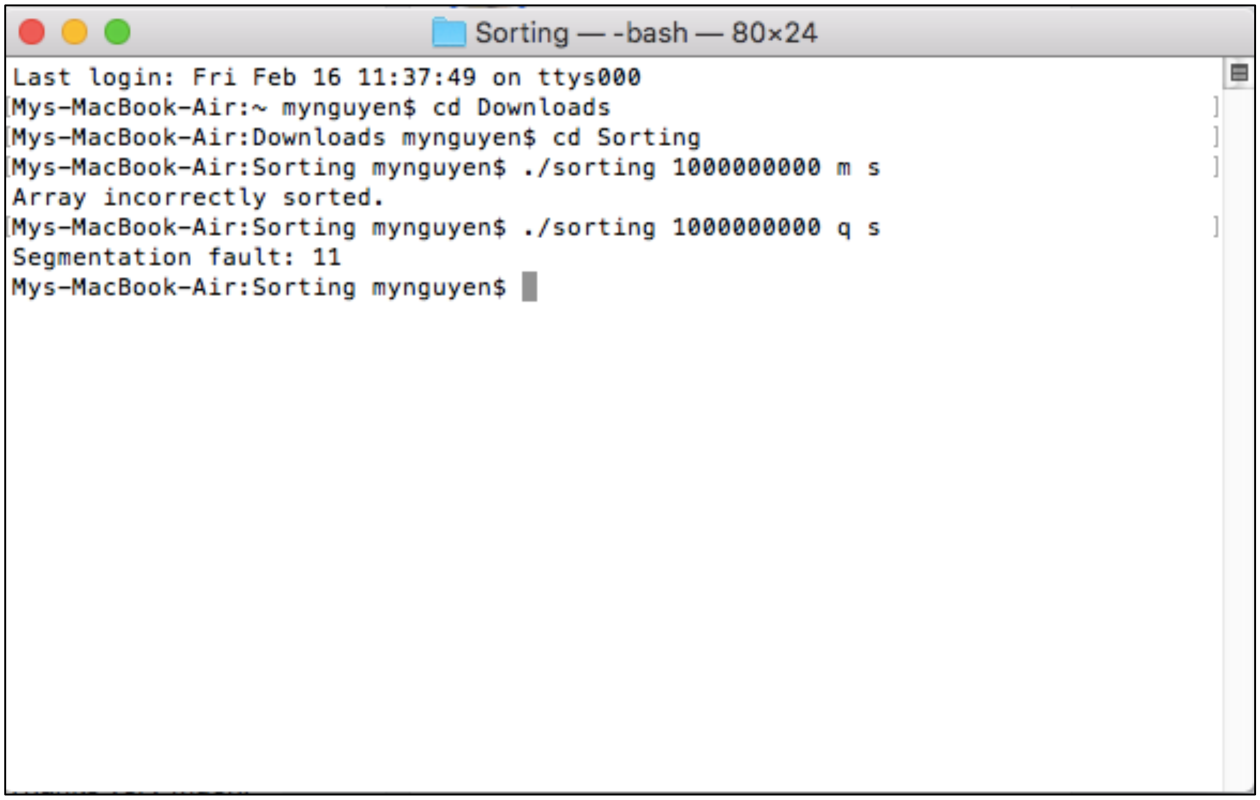
4.1 Found Issues and Solutions

The first issue deals with not having sufficient stack space for testing certain algorithms and its associated types. A segmentation fault was returned and can be seen in the figures below.



```
Attempt 3:         0 ms
Median time:            0 ms
Mys-MacBook-Air:Sorting-2 mynguyen$ ./sorting 10000 q c
Array successfully sorted.
Array successfully sorted.
Array successfully sorted.
Attempt 1:        72 ms
Attempt 2:        75 ms
Attempt 3:        71 ms
Median time:           72 ms
Mys-MacBook-Air:Sorting-2 mynguyen$ ./sorting 100000 q c
Array successfully sorted.
Array successfully sorted.
Array successfully sorted.
Attempt 1:      7124 ms
Attempt 2:      7078 ms
Attempt 3:      7037 ms
Median time:          7078 ms
Mys-MacBook-Air:Sorting-2 mynguyen$ ./sorting 1000000 q c
Segmentation fault: 11
Mys-MacBook-Air:Sorting-2 mynguyen$ ./sorting 900000 q c
Segmentation fault: 11
Mys-MacBook-Air:Sorting-2 mynguyen$ ./sorting 8800000 q c
```

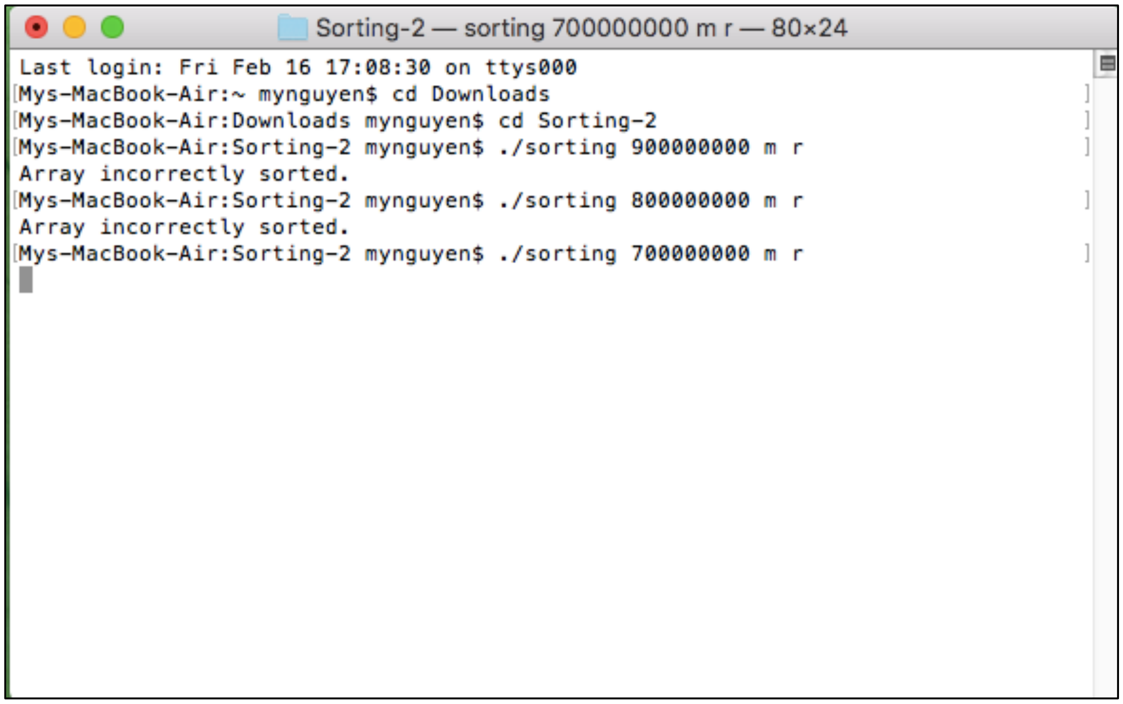
Figure 1: Quick Sort Constant Segmentation Fault



```
Last login: Fri Feb 16 11:37:49 on ttys000
Mys-MacBook-Air:~ mynguyen$ cd Downloads
Mys-MacBook-Air:Downloads mynguyen$ cd Sorting
Mys-MacBook-Air:Sorting mynguyen$ ./sorting 1000000000 m s
Array incorrectly sorted.
Mys-MacBook-Air:Sorting mynguyen$ ./sorting 1000000000 q s
Segmentation fault: 11
Mys-MacBook-Air:Sorting mynguyen$
```

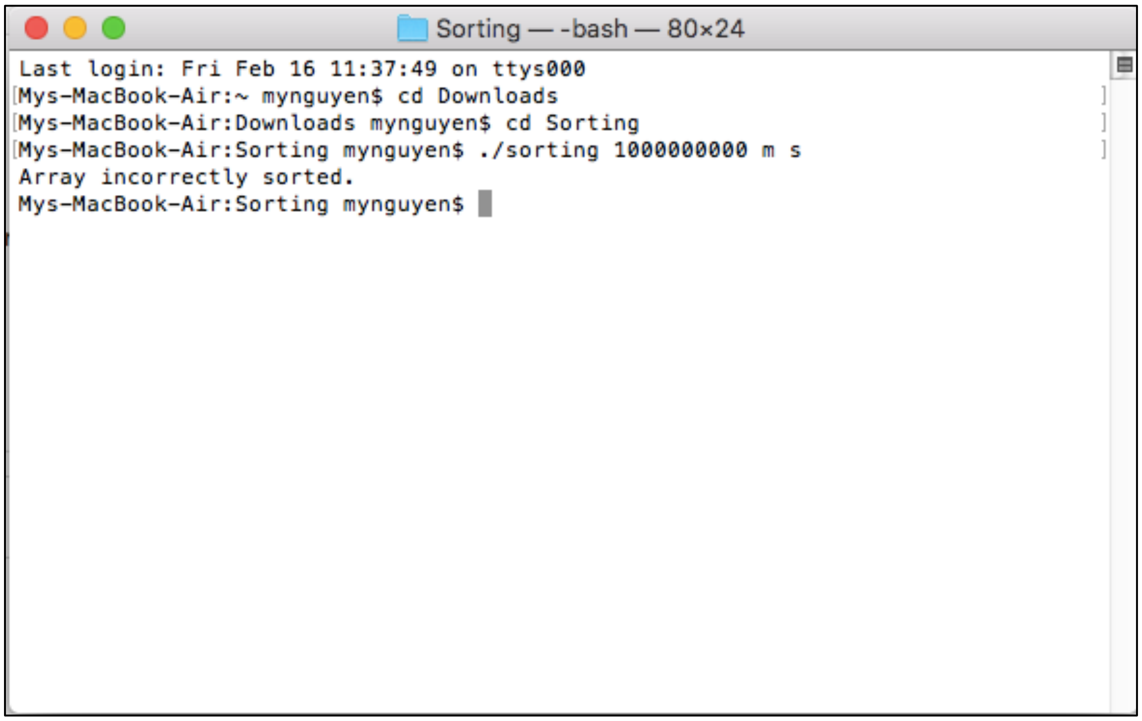
Figure 2: Quick Sort Sorted Segmentation Fault

The second issue deals with incorrectly sorted arrays seen in the figures below. This issue is possible due to the large array size and insufficient stack space.



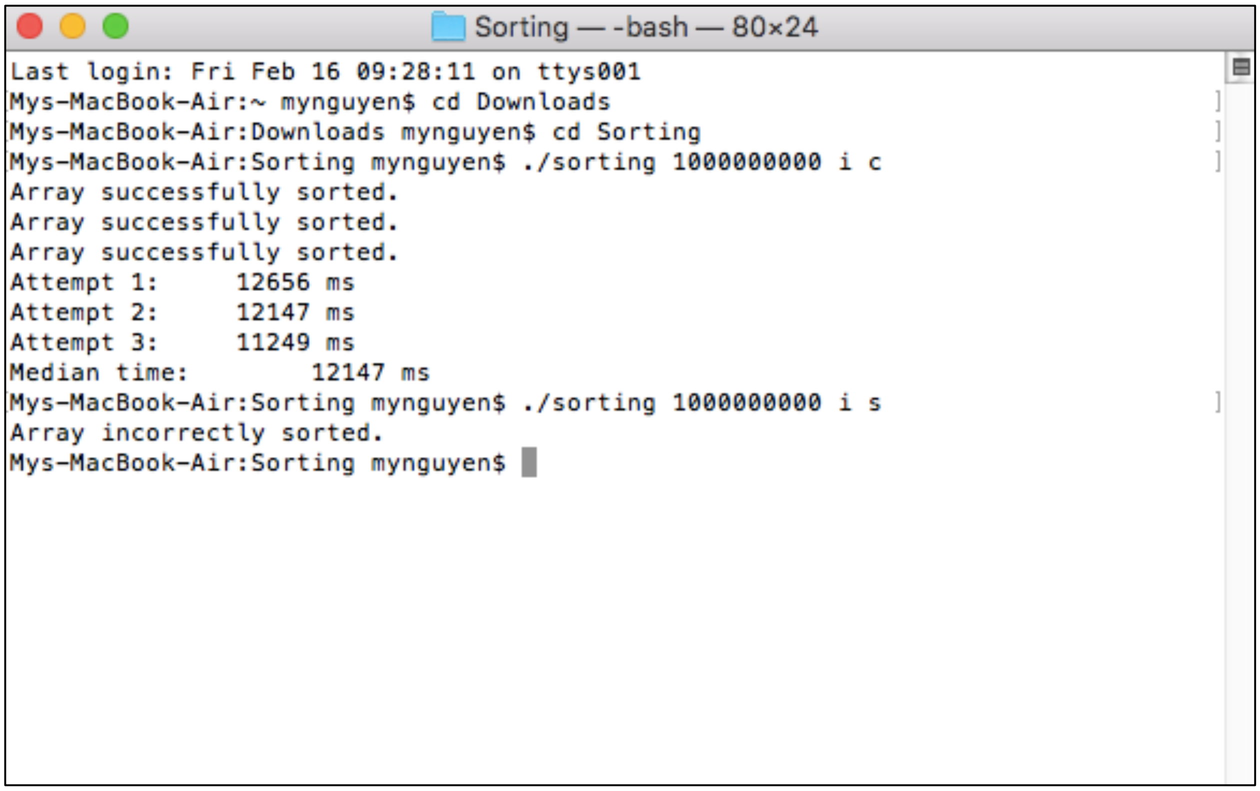
```
Sorting-2 — sorting 700000000 m r — 80×24
Last login: Fri Feb 16 17:08:30 on ttys000
Mys-MacBook-Air:~ mynguyen$ cd Downloads
Mys-MacBook-Air:Downloads mynguyen$ cd Sorting-2
Mys-MacBook-Air:Sorting-2 mynguyen$ ./sorting 900000000 m r
Array incorrectly sorted.
Mys-MacBook-Air:Sorting-2 mynguyen$ ./sorting 800000000 m r
Array incorrectly sorted.
Mys-MacBook-Air:Sorting-2 mynguyen$ ./sorting 700000000 m r
```

Figure 3: Merge Sort Random Incorrectly Sorted Array



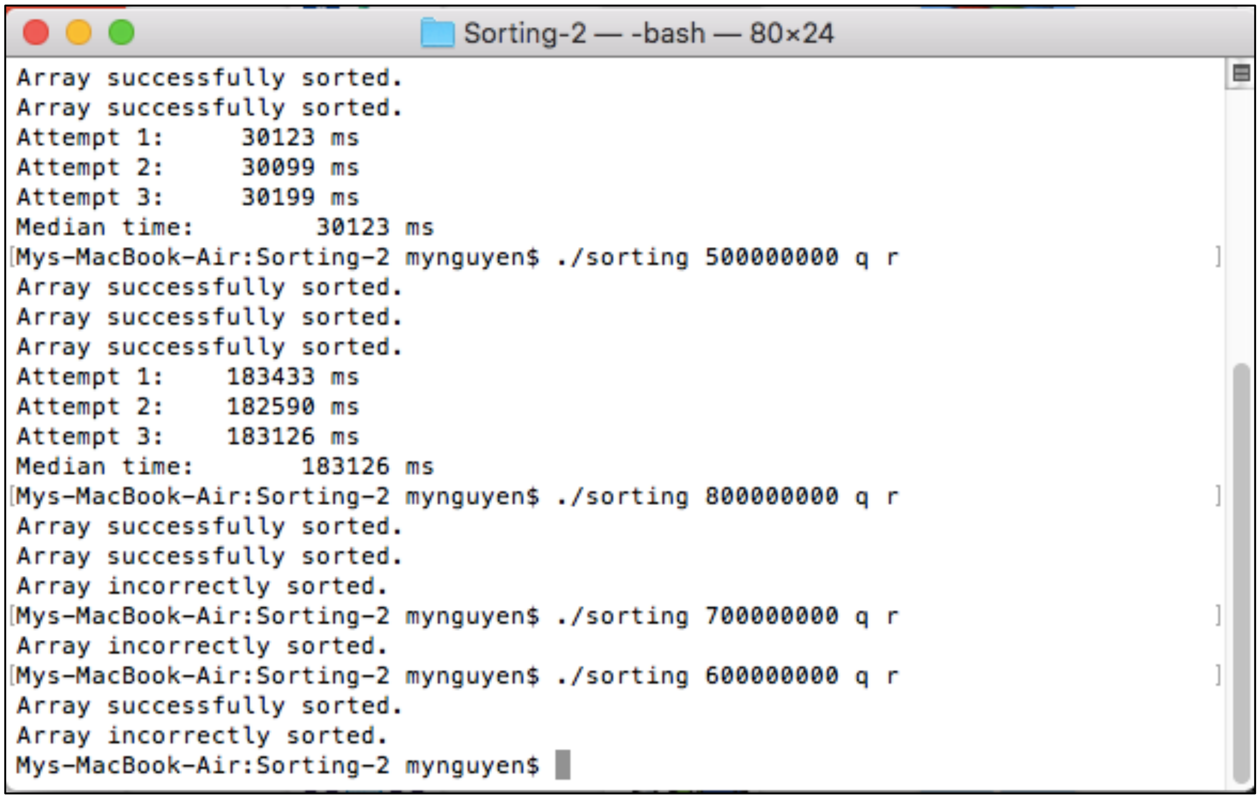
```
Sorting — -bash — 80×24
Last login: Fri Feb 16 11:37:49 on ttys000
Mys-MacBook-Air:~ mynguyen$ cd Downloads
Mys-MacBook-Air:Downloads mynguyen$ cd Sorting
Mys-MacBook-Air:Sorting mynguyen$ ./sorting 1000000000 m s
Array incorrectly sorted.
Mys-MacBook-Air:Sorting mynguyen$
```

Figure 2: Merge Sort Sorted Incorrectly Sorted Array



```
Last login: Fri Feb 16 09:28:11 on ttys001
Mys-MacBook-Air:~ mynguyen$ cd Downloads
Mys-MacBook-Air:Downloads mynguyen$ cd Sorting
Mys-MacBook-Air:Sorting mynguyen$ ./sorting 1000000000 i c
Array successfully sorted.
Array successfully sorted.
Array successfully sorted.
Attempt 1:     12656 ms
Attempt 2:     12147 ms
Attempt 3:     11249 ms
Median time:        12147 ms
Mys-MacBook-Air:Sorting mynguyen$ ./sorting 1000000000 i s
Array incorrectly sorted.
Mys-MacBook-Air:Sorting mynguyen$
```

Figure 2: Insertion Sort Sorted Incorrectly Sorted Array



```
Array successfully sorted.
Array successfully sorted.
Attempt 1:     30123 ms
Attempt 2:     30099 ms
Attempt 3:     30199 ms
Median time:        30123 ms
Mys-MacBook-Air:Sorting-2 mynguyen$ ./sorting 500000000 q r
Array successfully sorted.
Array successfully sorted.
Array successfully sorted.
Attempt 1:    183433 ms
Attempt 2:    182590 ms
Attempt 3:    183126 ms
Median time:       183126 ms
Mys-MacBook-Air:Sorting-2 mynguyen$ ./sorting 800000000 q r
Array successfully sorted.
Array successfully sorted.
Array incorrectly sorted.
Mys-MacBook-Air:Sorting-2 mynguyen$ ./sorting 700000000 q r
Array incorrectly sorted.
Mys-MacBook-Air:Sorting-2 mynguyen$ ./sorting 600000000 q r
Array successfully sorted.
Array incorrectly sorted.
Mys-MacBook-Air:Sorting-2 mynguyen$
```

Figure 2: Quick Sort Random Incorrectly Sorted Array

The solution for these two issues involve running *ulimit -s unlimited* in Linux to increase the available stack space to run large array sizes i.e. one billion. In addition to experiencing limited stack sizes, the stack size needed to be changed for every run because it's based on per shell. Overall, the behavior are as expected and similar to the results found in class.