Tyler Simoni
COT4400
6/19/2017

# Project 1: Asymptotic Analysis of Sorting Algorithms

## 1. Results

### 1.1. Selection Sort

Selection Sort works by dividing the array into two parts: 1) A subarray of already sorted elements and 2) a subarray of elements yet to be sorted. Moving left to right, the algorithm finds the smallest (or largest) element in the unsorted subarray, swaps it with leftmost unsorted element, and then moves the subarray bounds by one element. In all cases (best, average, and worst), selection has a complexity of $\Theta(n^2)$. This makes it not particularly useful for large sets of data.

The tested results for Selection sort on constant, sorted, and random data are as follows:

| | $n_{min}$ | $t_{min}$ | $n_{max}$ | $t_{max}$ |
|---|---|---|---|---|
| SC | 5000 | 15 | 750000 | 521187 |
| SS | 5000 | 15 | 750000 | 525093 |
| SR | 5000 | 31 | 750000 | 524578 |

### 1.2. Insertion Sort

Insertion Sort works by picking one element for each iteration of the loop, locating where it belongs in the sorted array of elements, inserts it, and repeats until all elements have been sorted. The best case for Insertion sort is for an array that is already sorted, yielding a complexity of $\Omega(n)$. The worst case is for an array in reverse order, which yields a complexity of $O(n^2)$. Average case is also $\Theta(n^2)$ which, like Selection sort, makes it bad for handling large data sets.

The tested results for Insertion sort on constant, sorted, and random data are as follows:

| | $n_{min}$ | $t_{min}$ | $n_{max}$ | $t_{max}$ |
|---|---|---|---|---|
| IC | 5000000 | 15 | 1000000000 | 2703 |
| IS | 5000000 | 15 | 1000000000 | 2718 |
| IR | 5000 | 15 | 1000000 | 528140 |

Tyler Simoni
COT4400
6/19/2017

## 1.3. Merge Sort

The divide-and conquer algorithm, Merge Sort, starts by repeatedly splitting the array in half until the base case of a single element is reached in each subarray. It then returns up, merging each subarray, sorting them, and merging further up until the original array of size n is reached, resulting in a fully sorted array. Merge sorts best, average, and worst case complexities are Θ (n lg n).

The tested results for Merge sort on constant, sorted, and random data are as follows:

| | $n_{min}$ | $t_{min}$ | $n_{max}$ | $t_{max}$ |
|---|---|---|---|---|
| MC | 750000 | 31 | 1000000000 | 71250 |
| MS | 750000 | 46 | 1000000000 | 74437 |
| MR | 100000 | 15 | 1000000000 | 192000 |

## 1.4. Quick Sort

The comparison sorting algorithm, Quick Sort, works by choosing a pivot value in the array. For the best case, a random element is chosen. It then moves everything smaller than the pivot to the left and everything bigger, to the right. It then continues doing this to each subarray until the entire array of size n is sorted. The best and average case for Quick Sort are Θ (n lg n). The worst case for Quick Sort is O ($n^2$); however, this is rare.

The tested results for Quick sort on constant, sorted, and random data are as follows:

| | $n_{min}$ | $t_{min}$ | $n_{max}$ | $t_{max}$ |
|---|---|---|---|---|
| QC | 10000 | 46 | n/a | n/a |
| QS | 500000 | 15 | 1000000000 | 53125 |
| QR | 100000 | 15 | 1000000000 | 192640 |

Note: The stack size was not sufficient enough to complete quicksort using constant data. The `ulimit` and `setrlimit` commands were used but were unable to increase the size of the stack.

Tyler Simoni
COT4400
6/19/2017

## 1.5. Complete Data

| Input | Selection Sort | | | Insertion Sort | | |
|---|---|---|---|---|---|---|
| | Constant | Sorted | Random | Constant | Sorted | Random |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 |
| 100 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5000 | 15 | 15 | 31 | 0 | 0 | 15 |
| 10000 | 93 | 93 | 93 | 0 | 0 | 46 |
| 50000 | 2328 | 2312 | 2328 | 0 | 0 | 1296 |
| 100000 | 9250 | 9265 | 9296 | 0 | 0 | 5203 |
| 500000 | 231531 | 231296 | 231312 | 0 | 0 | 130343 |
| 750000 | 521187 | 525093 | 524578 | 0 | 0 | 293484 |
| 1000000 | 926640 | 934031 | 933000 | 0 | 0 | 528140 |
| 5000000 | >1000000 | >1000000 | >1000000 | 15 | 15 | >1000000 |
| 10000000 | >1000000 | >1000000 | >1000000 | 31 | 31 | >1000000 |
| 50000000 | >1000000 | >1000000 | >1000000 | 140 | 140 | >1000000 |
| 100000000 | >1000000 | >1000000 | >1000000 | 281 | 265 | >1000000 |
| 500000000 | >1000000 | >1000000 | >1000000 | 1359 | 1359 | >1000000 |
| 1000000000 | >1000000 | >1000000 | >1000000 | 2703 | 2718 | >1000000 |

| Input | Merge Sort | | | Quick Sort | | |
|---|---|---|---|---|---|---|
| | Constant | Sorted | Random | Constant | Sorted | Random |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 |
| 100 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5000 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10000 | 0 | 0 | 0 | 46 | 0 | 0 |
| 50000 | 0 | 0 | 0 | 968 | 0 | 0 |
| 100000 | 0 | 0 | 15 | 3890 | 0 | 15 |
| 500000 | 0 | 0 | 62 | n/a | 15 | 62 |
| 750000 | 31 | 46 | 93 | n/a | 31 | 93 |
| 1000000 | 46 | 46 | 125 | n/a | 31 | 125 |
| 5000000 | 281 | 281 | 718 | n/a | 203 | 718 |
| 10000000 | 562 | 578 | 1500 | n/a | 421 | 1531 |
| 50000000 | 3062 | 3187 | 8156 | n/a | 2296 | 8312 |
| 100000000 | 6312 | 6718 | 16937 | n/a | 4765 | 17234 |
| 500000000 | 34796 | 36187 | 92984 | n/a | 25437 | 93453 |
| 1000000000 | 71250 | 74437 | 192000 | n/a | 53125 | 192640 |

Tyler Simoni
COT4400
6/19/2017

# 2. Analysis

## 2.1. Selection Sort

The tested complexity for Selection Sort is calculated as:

| | $t_{max}/t_{min}$ | n ratio | n lg n ratio | $n^2$ ratio | Behavior |
|---|---|---|---|---|---|
| SC | 34745.8 | 150 | 238.2444812 | 22500 | $n^2$ |
| SS | 35006.2 | 150 | 238.2444812 | 22500 | $n^2$ |
| SR | 16921.871 | 150 | 238.2444812 | 22500 | $n^2$ |

The experimental results match up exactly to the theoretical results. This was expected as the data sets have to be fairly small in order to achieve decent complexity times. Once you cross 1 million input elements, the time to sort can be 15 minutes or more.

## 2.2. Insertion Sort

The tested complexity for Insertion Sort is calculated as:

| | $t_{max}/t_{min}$ | n ratio | n lg n ratio | $n^2$ ratio | Behavior |
|---|---|---|---|---|---|
| IC | 180.2 | 200 | 268.6980236 | 40000 | n |
| IS | 181.2 | 200 | 268.6980236 | 40000 | n |
| IR | 35209.333 | 200 | 324.414634 | 40000 | $n^2$ |

The experimental results nearly match up to the theoretical analysis of the Insertion Sort algorithm. The only difference is with the sorted elements being O(n) instead of O($n^2$). This could have been caused by a number of factors including number of currently running processes and CPU speed.

## 2.3. Merge Sort

The tested complexity for Merge Sort is calculated as:

| | $t_{max}/t_{min}$ | n ratio | n lg n ratio | $n^2$ ratio | Behavior |
|---|---|---|---|---|---|
| MC | 34745.8 | 150 | 238.2444812 | 22500 | n lg n |
| MS | 35006.2 | 150 | 238.2444812 | 22500 | n lg n |
| MR | 16921.871 | 150 | 238.2444812 | 22500 | n lg n |

The experimental results match up exactly to the theoretical analysis of Merge Sort. This algorithm is one of the more complex, but effective sorting algorithms. The only thing to really worry about is very very large data sets and the amount of memory available in the system.

Tyler Simoni
COT4400
6/19/2017

## 2.4. Quick Sort

The tested complexity for Quick Sort is calculated as:

| | $t_{max}/t_{min}$ | n ratio | n lg n ratio | $n^2$ ratio | Behavior |
|---|---|---|---|---|---|
| QC | n/a | n/a | n/a | n/a | n/a |
| QS | 3541.6667 | 2000 | 3158.465475 | 4000000 | n lg n |
| QR | 12842.667 | 10000 | 18000 | 100000000 | n |

The experimental results do not match up to the theoretical results. As mentioned before, the constant data was unable to be obtained as the stack size was not sufficient. This caused a segmentation fault in the program due to a stack overflow. The stack has the ability to be changed; however, the system used did not allow it. The random data yielded a linear behavior, which is incorrect as the expected result is $O(n^2)$. This could be caused by the data sets being incorrectly sized or a miscalculation of the final results. The average experimental behavior did match the theoretical analysis.