Asymptotic Analysis Project
COT 4400, Spring 2019
Julie Reyes

This report evaluates the empirical run time of four sorting algorithms and compares those times to the theoretical complexities for each algorithm. Table 1 demonstrates the data collected from running each of the four algorithms with constant, random, and sorted arrays of different sizes denoted by n. Table 2 demonstrates the theoretical growth complexity based on the ratio of $f(n_{max})/f(n_{min})$ and compares those functions to the observed time ratio t_{max}/t_{min} to the theoretical growth complexities to determine which behavior most closely approximates the observed behavior. Selection sort performed with quadratic behavior for arrays of constant, random, and sorted elements with very little variation. This observation correlated closely to the quadratic theoretical complexity that describes SectionSort. InsertionSort exhibited linear time complexity for arrays of sorted and constant elements. This observation corresponds to the theoretical best-case complexity of InsertionSort. This is due to the algorithm not needing to shift elements in the array to insert other values. InsertionSort with random elements, on the other hand, performed at quadratic complexity, which is the theoretical average and worst-case complexity for InsertionSort. This is caused by the need to insert elements in the array and shift other elements, creating quadratic run times in arrays that are sufficiently large. MergeSort empirically performed at approximately $\Theta(nlg(n))$ time complexity. Specifically, the algorithm performs slightly better than $\Theta(nlg(n)$ with sorted and constant arrays. This is most likely due to not being able to select a n_{max} sufficiently large enough to properly analyze the growth rate of the algorithm because of the inherent program restrictions limiting the maximum size of n. Quicksort performed at approximately quadratic time complexity when all the values of the array where constant. This time complexity corresponds to the theoretical worst-case time for QuickSort. This is most likely due to the choice of the pivot in a constant element array causing more recursive calls than a sorted or random element array. It is important to note that to properly test QuickSort, the size of the stack on the machine running the sorting program had to be increased to allow for the increase in recursive calls in large arrays with constant elements. This relates to the theoretical worst case scenario for QuickSort as it creates an approximate recursion expression of $T(n) = T(n-1) + \Theta(1)$, which is of quadratic time complexity. QuickSort for arrays of sorted and random elements preformed empirically at $\Theta(nlg(n))$ time complexity, which is equivalent to the theoretical average and best-case complexity. Thus, for sorted and random element arrays, the choice of pivot was more optimal than in constant element arrays. Empirically the four sorting algorithms tested in this project approximated the theoretical time complexities discussed in class. Differences in performance of the same algorithm was dependent of the contents of the arrays in some cases. This was due to the inherent properties of the algorithms. For example, QuickSort performed poorly with constant element arrays and InsertionSort

performed poorly with random element arrays. In general, the empirical evidence supports the theoretical time complexities discussed in class for these sorting algorithm

Table 1

	n_{max}	t_{max}	n_{min}	t_{min}
SS	800000	557388	10000	97
SR	800000	551945	10000	97
SC	800000	550085	10000	96
IS	1000000000	3318	10000000	38
IR	1000000	548551	10000	62
IC	1000000000	3322	10000000	37
MS	1000000000	91095	300000	22
MR	1000000000	221727	150000	24
MC	1000000000	84818	300000	20
QS	1000000000	67260	400000	22
QR	1000000000	249017	140000	29
QC	1000000	467868	6000	21

Table 2

	t_{max}/t_{min}	n ratio	n lg(n)	n^2	Behavior
SS	5746	80	118	6400	n^2
SR	5690	80	118	6400	n^2
SC	5730	80	118	6400	n^2
IS	87	100	129	10000	n
IR	8848	100	150	10000	n^2
IC	90	100	129	10000	n
MS	4141	3333	5477	11111111	nln(n)
MR	9239	6667	11592	44444444	nln(n)
MC	4241	3333	5477	11111111	nln(n)
QS	3057	2500	4016	6250000	nln(n)
QR	8587	7143	12492	51020408	nln(n)
QC	22279	167	265	27778	n^2