Sara Savitz

U37713110

Analysis of Algorithms

Project 1

For this project, we must analyze the behavior of four sorting algorithms, including Selection Sort, Insertion Sort, Merge Sort, and Quick Sort. All four algorithms were tested using a constant array, a random array, and a sorted array. Our goal is to compare the empirical time complexity of each algorithm with their expected time complexity. The table below shows a collection of empirical data gathered in a constant computing environment; this data includes the various minimum and maximum numbers of elements being sorted, as well as the time taken to sort them.

Empirical Data

	Nmin	Tmin	Nmax	Tmax
SC	10^4	140ms	10^6	14453ms
SR	10^4	156ms	10^6	14593ms
SS	10^4	140ms	10^6	14578ms
IC	10^7	31ms	10^9	4328ms
IR	10^4	78ms	10^5	8484ms
IS	10^7	62ms	10^9	4312ms
MC	10^6	78ms	10^9	126109ms
MR	10^6	203ms	10^8	25390ms
MS	10^6	62ms	10^8	10500ms
QC	10^4	78ms	10^5	6375ms
QR	10^6	203ms	10^8	26093ms
QS	10^6	46ms	10^7	7468ms

Expected Behavior

	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	$\Omega(n \lg n)$	$\Theta(n \lg n)$	$O(n^2)$

The time complexities for each algorithm were calculated and compared to the theoretical time complexities. These values were derived from the minimum and maximum values gathered during the experiment.

Actual Behavior

Algorithm	Tmax/Tmin	F1 = n	F2 = nlog(n)	F3	Behavior
SC	103	10	13	100	n^2
SR	94	10	13	100	n^2
SS	104	10	13	100	n^2
IC	140	100	129	10000	nlg(n)
IR	109	10	13	100	n^2
IS	70	100	129	10000	n
MC	1617	1000	1500	1000000	nlg(n)
MR	125	100	133	10000	nlg(n)
MS	169	100	133	10000	nlg(n)
QC	82	10	13	1000	nlg(n)
QR	129	100	133	10000	nlg(n)
QS	162	10	12	100	n^2

Over the course of the project, there were a few instances worth mentioning where actual results did not match up to the expected results. Let us examine each algorithm individually:

For Selection Sort, all behavior observed was the same as the expected behavior. While Selection Sort has proven to be the least efficient algorithm, it should be noted that it has the least variance in time complexity when presented with different types of array values (constant, random, sorted). This makes it a reliable sorting algorithm for small amounts of data that may differ significantly in content.

For Insertion Sort, our actual results were slightly different than expected results. All three types of behavior were exhibited on the three types of array values, with worst-case = O(n^2), and best-case Ω(nlg(n)), as opposed to Ω(n). The best-case performance was observed using an array of constant values, which makes sense because no data needed to be shifted or inserted, significantly reducing the most time-consuming aspects of Selection Sort.

For Merge Sort, all three instances of the algorithm behaved as expected, with time complexity Θ(nlg(n)). Merge Sort performed the best of all four algorithms. Not only did it have the fastest time complexity in all three cases, it also was able to sort the largest amounts of elements out of the bunch. This makes sense because it is a recursive sorting algorithm, which tackle large chunks of data very quickly. It can be inferred through observation that Merge Sort's performance is seemingly unaffected by the type of values contained in the array to be sorted.

For Quick Sort, all behavior observed was exactly as expected. Contrary to its name, Quick Sort was not the most efficient algorithm of the four, ranking at about second place (with Insertion Sort at a very close third). Surprisingly, an array of constant

size took the longest at values exceeding 10^5, making this instance of the algorithm the least efficient, despite its Θ(nlg(n)) time complexity due to recursive behavior. This is most likely a result of the pivot aspect of the algorithm, which accomplishes very little in the presence of constant values. Based on these observations, it is strongly advised to not use Quick Sort to sort arrays containing constant values. Despite this factor, Quick Sort did appear to work very well on arrays containing random values. It is clear that Quick Sort's performance depends greatly on the type of information being sorted.