Analysis of Sorting Algorithms

Mark Philipp

Isaac Fielding

California State University Fresno

March 9, 2018

Introduction

This report attempts to analyze common sorting algorithms and provide conclusions about the time complexity of each algorithm. The goal is to visualize the running times of sorting algorithms when they are used on arrays of specific sizes.

Methodology

Five sorting algorithms were tested: bubble sort, insertion sort, selection sort, quick sort, and merge sort. By using this variety of sorting algorithms, we were able to get data and draw conclusions for several O( algorithms and two O(NLogN) algorithms.

To accurately measure time complexity, four different types of sorted arrays were used. Arrays were either filled entirely with random values, ordered, reversed, or ordered with a shuffle applied to 10% of the values. The reason this approach was used is because sorting algorithms will have different running times depending on the initial sorting of an array they are operating on. For instance, if an array is already ordered, then most sorting algorithms will finish operating on the array quickly. In contrast, an array that is initially reverse ordered will take a long time for a sorting algorithm to operate on it. Additionally, each array utilized positive integer values for their data types.

For every type of array, five different array sizes were tested. Arrays were tested at sizes 10, 1000, 10000, 100000, 1000000. The exponentially large sizes for the arrays should clearly demonstrate the time complexity of the different arrays. O( sorting algorithms should definitely show an exponentially slower running time as the size of the arrays increase.

Each type of array of each size ran each sorting algorithm 100 times. For instance, a randomly shuffled array will run bubble sort 100 times at size 10, 100 times at size 1000, 100 times at size 10000, 100 times at size 100000, and 100 times at size 1000000. It’s important to note that the data contained in the array was the same every iteration. This means that a randomly ordered array of size 10 contained the same exact values in the same exact indexes on iteration 1 as it did on iteration 100. This approach ensures that the algorithm is doing the same exact operations per iteration. Additionally, by running 100 times each, we are able to get a more precise average running time for each algorithm.

Time was measured in seconds using the C++ chrono library from the standard template library. The timer would start at the beginning of each iteration and would take a final running time after the sorting algorithm had finished sorting the given array. It’s important to note that an algorithm execution would abort if a single iteration took longer than 5 minutes to finish sorting. Additionally, an entire benchmark, which contains 100 iterations, would abort if the total running time exceeded 2 hours. These limitations were implemented due to the running times of the O( algorithms at size 1000000. These algorithms take exceptionally long to finish executing on such large arrays. To finish the benchmarks within a reasonable time, the time limits had to be implemented. This cutoff time will be reflected in the data charts.

Results

Bubble Sort

The below graph shows the results of using bubble sort on multiple types of arrays. Time is in seconds.