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 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 can 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.

All benchmarks were run on a desktop computer running Windows 10 64-bit with the following specs:

* Intel core i7 quad-core CPU 3.5 GHz, 8 MB L3 cache
* 16 GB DDR3 PC3 14900 RAM
* Nvidia GeForce GTX 1060 6GB

Results

Graph 1

Running time of bubble sort in relation to the size of an array.

Table 1-1

Total average running times of bubble sort algorithm on different types of arrays of variable size.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Bubble Sort Total Average Running Times** | | | | |
| **Size** | **Random** | **Ordered** | **Reverse** | **Shuffled** |
| 10 | 0.000038 s | 0.000005 s | 0.000044 s | 0.000025 s |
| 1000 | 0.382189 s | 0.000425 s | 0.411766 s | 0.385060 s |
| 10000 | 40.460981 s | 0.004100 s | 46.768684 s | 43.140773 s |
| 100000 | 105.198171 s | 0.000310 s | 143.151126 s | 12.099275 s |
| 1000000 | > 300 s | 0.395795 s | > 300 s | > 300 s |

Table 1-2

Standard deviation of bubble sort algorithm on different types of arrays of variable size

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Bubble Sort Standard Deviation** | | | | |
| **Size** | **Random** | **Ordered** | **Reverse** | **Shuffled** |
| 10 | 0.000001 s | 0.000001 s | 0.000006 s | 0.000001 s |
| 1000 | 0.002541 s | 0.000019 s | 0.001697 s | 0.001961 s |
| 10000 | 0.865294 s | 0.000106 s | 6.334522 s | 6.209309 s |
| 100000 | 5.258011 s | 0.000038 s | 11.508834 s | 0.121674 s |
| 1000000 | 0.000195 s | 0.000949 s | 0.000184 s | 0.003572 s |

Bubble Sort Analysis

The above data shows what one would expect from bubble sort. As the size of the array increases exponentially, so does the running time. This is a clear example of O( time complexity. Worth noting is the time it takes to sort an ordered array. Bubble sort’s best-case time complexity is O(N) and it appears to show on the ordered array. This is likely because bubble sort quits running if it doesn’t do a swap in a single iteration. Upon the first traversal of the array, bubble sort will know it is sorted and will finish in a fast time.

Another point worth mentioning is bubble sort’s worst-case running time occurs with the reverse ordered array. This is because every element of the array is out of order, which means bubble sort will perform swapping operations on every element during every iteration. This rapidly increases the running time of the algorithm. The reverse ordered array clearly shows the downside of using an O( algorithm on reverse ordered data.

One of the more interesting pieces of data is the average running time of the shuffled array at size 100000. There is a clear dip in the running time where we would normally expect an obvious increase in running time when compared with size 10000. The most likely explanation for this anomaly can be found in the implementation of the shuffle array. The shuffle array is first created as an entirely ordered array. Once that is done, it has 10% of its values reorganized using the rand function from the C++ STL to select random indexes to swap. Due to the way random number generators work and the time-based seed that was being used, it is possible that the array was shuffled in such a way that the array was still mostly ordered. It is also possible the shuffling ended up causing the front 10% of the array to be unsorted but left the back 90% untouched. This would cause bubble sort to start slowly and finish rather quickly once the first 10% is finished sorting.

Graph 2

Running time of insertion sort in relation to the size of an array

Table 2-1

Total average running times of insertion sort algorithm on different types of arrays of variable size.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Insertion Sort Total Average Running Times** | | | | |
| **Size** | **Random** | **Ordered** | **Reverse** | **Shuffled** |
| 10 | 0.000012 s | 0.000005 s | 0.000024 s | 0.000009 s |
| 1000 | 0.108688 s | 0.000427 s | 0.216164 s | 0.035311 s |
| 10000 | 10.474407 s | 0.004118 s | 21.165292 s | 3.187619 s |
| 100000 | 72.722735 s | 0.000222 s | 136.800525 s | 4.987270 s |
| 1000000 | > 300 s | 0.396520 s | > 300 s | > 300 s |

Table 2-2

Standard deviation of insertion sort algorithm on different types of arrays of variable size

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Insertion Sort Standard Deviation** | | | | |
| **Size** | **Random** | **Ordered** | **Reverse** | **Shuffled** |
| 10 | 0.000000171 s | 0.000000995 s | 0.000002288 s | 0.000000174 s |
| 1000 | 0.000996382 s | 0.000015392 s | 0.005828847 s | 0.000395228 s |
| 10000 | 0.018254461 s | 0.000127610 s | 0.470253577 s | 0.007718826 s |
| 100000 | 5.530829562 s | 0.000000716 s | 2.301054098 s | 0.200565331 s |
| 1000000 | 0.000217083 s | 0.001691333 s | 0.000229612 s | 0.000250716 s |

Insertion Sort Analysis

The data above reflects insertion sort’s average-case running time of O( complexity. As the size of the array exponentially increases, so does the running time of the algorithm. The exception is, again, the ordered array. Much like bubble sort, insertion sort can verify the ordering of an ordered array incredibly fast.

Much like bubble sort, the worse-case running time is represented by the reverse ordered array. This is because insertion sort will have to do swaps and comparisons to sort a reversed array. It’s worth noting that the worst-case insertion sort running time is faster than bubble sort’s worst-case running time.

The shuffled array performance is interesting because it is fast up until size 1000000. With bubble sort, the shuffled array performance more closely matched the random array performance. With insertion sort, however, the shuffled array is much closer to the ordered array. Again, this could be an anomaly caused by the randomness of the shuffle, which could cause the unsorted part of the array to cluster in the front 10% of the array.

Graph 3

Running time of selection sort in relation to the size of an array

Table 3-1

Total average running times of selection sort algorithm on different types of arrays of variable size.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Selection Sort Total Average Running Time** | | | | |
| **Size** | **Random** | **Ordered** | **Reverse** | **Shuffled** |
| 10 | 0.000008 s | 0.000007 s | 0.000007 s | 0.000008 s |
| 1000 | 0.001628 s | 0.001552 s | 0.001561 s | 0.001557 s |
| 10000 | 0.118933 s | 0.117583 s | 0.112293 s | 0.118529 s |
| 100000 | 10.443775 s | 11.227691 s | 9.914354 s | 10.437321 s |
| 1000000 | > 300 s | > 300 s | > 300 s | > 300 s |

Table 3-2

Standard deviation of selection sort algorithm on different types of arrays of variable size.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Selection Sort Standard Deviation** | | | | |
| **Size** | **Random** | **Ordered** | **Reverse** | **Shuffled** |
| 10 | 0.000004 s | 0.000001 s | 0.000001 s | 0.000003 s |
| 1000 | 0.000087 s | 0.000084 s | 0.000153 s | 0.000060 s |
| 10000 | 0.001460 s | 0.001098 s | 0.001510 s | 0.002099 s |
| 100000 | 0.020544 s | 0.387844 s | 0.031631 s | 0.018334 s |
| 1000000 | 0.062053 s | 0.000257 s | 0.000258 s | 0.000226 s |

Selection Sort Analysis

The results of selection sort differ drastically from the other O( algorithms. The data is so strange that we were inclined to think there was a problem with the implementation of the algorithm. After triple-checking the algorithm and ensuring correctness, we ran the benchmarks several more times and continued to get the same results. One possible explanation for the comparatively fast running times is the fact that selection sort’s average-case complexity is O( comparison but only O(N) swaps. Insertion sort, for instance, on average runs O( comparisons and swaps, which makes insertion sort, potentially, exponentially worse for running time.

Another unexpected result is that the running times across each type of array are very close to each other. With insertion sort and bubble sort, the ordered arrays took very little time at every size. With selection sort, however, this is not true. This data would suggest that selection sort is not ideal for sorting ordered arrays. One possible explanation for this is that selection sort does not have a condition to exit the algorithm early. Selection sort must compare times no matter what, which can lead to a longer running time for ordered arrays. However, selection sort clearly runs faster than insertion and bubble sort for almost every other case.

Graph 4

Running time of merge sort in relation to the size of an array

Table 4-1

Total average running times of merge sort algorithm on different types of arrays of variable size.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Merge Sort Total Average Running Time** | | | | |
| **Size** | **Random** | **Ordered** | **Reverse** | **Shuffled** |
| 10 | 0.000004 s | 0.000003 s | 0.000003 s | 0.000004 s |
| 1000 | 0.000452 s | 0.000392 s | 0.000383 s | 0.000386 s |
| 10000 | 0.003527 s | 0.003000 s | 0.002890 s | 0.003063 s |
| 100000 | 0.047896 s | 0.040697 s | 0.040462 s | 0.041826 s |
| 1000000 | 0.497826 s | 0.420809 s | 0.421410 s | 0.427398 s |

Table 4-2

Standard deviation of merge sort algorithm on different types of arrays of variable size.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Merge Sort Standard Deviation** | | | | |
| **Size** | **Random** | **Ordered** | **Reverse** | **Shuffled** |
| 10 | 0.000002 s | 0.000001 s | 0.000001 s | 0.000002 s |
| 1000 | 0.000057 s | 0.000054 s | 0.000051 s | 0.000050 s |
| 10000 | 0.000175 s | 0.000250 s | 0.000136 s | 0.000213 s |
| 100000 | 0.001408 s | 0.000337 s | 0.000235 s | 0.000267 s |
| 1000000 | 0.004608 s | 0.001637 s | 0.001718 s | 0.001911 s |

Merge Sort Analysis

Merge sort’s data clearly shows the benefits of using divide and conquer sorting algorithms. Even at an array size of 1000000, merge sort finishes sorting in less than a second on average. In comparison, the O(, algorithms would frequently exceed our maximum limit of 300 seconds. Note that the graph only shows a maximum of 0.5 seconds.

One interesting data point is that merge sort appeared to perform the worst on randomized arrays. With all the O( algorithms, the reverse array would exhibit the worst-case running time. With merge sort, the worst case running time appears to be from randomized arrays. This is likely because of how merge sort performs the merging process. With a random array, merge sort runs the risk of performing the maximum number of comparisons possible. In contrast, the reverse array will only perform half the number of maximum comparisons because the numbers are reversed. When a swap occurs between the left and right half of the array, it will guarantee the correct final position of the values being swapped.

One potential downside for merge sort is the possibility of stack overflows. Because the algorithm is recursive, it allocates a large amount of memory on the stack. We ran into this issue during our benchmarking of sizes 10000 and higher. However, this issue was quickly fixed by properly de-allocating memory after every benchmark iteration.

Graph 5

Running time of quick sort in relation to the size of an array

Table 5-1

Total average running times of quick sort algorithm on different types of arrays of variable size.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Quick Sort Total Average Running Time** | | | | |
| **Size** | **Random** | **Ordered** | **Reverse** | **Shuffled** |
| 10 | 0.000002 s | 0.000002 s | 0.000002 s | 0.000002 s |
| 1000 | 0.000175 s | 0.000080 s | 0.000090 s | 0.000088 s |
| 10000 | 0.002192 s | 0.000866 s | 0.000954 s | 0.001103 s |
| 100000 | 0.002192 s | 0.000866 s | 0.000954 s | 0.001103 s |
| 1000000 | 0.295504 s | 0.103080 s | 0.112400 s | 0.119805 s |

Table 5-2

Total average running times of quick sort algorithm on different types of arrays of variable size.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Quick Sort Standard Deviation** | | | | |
| **Size** | **Random** | **Ordered** | **Reverse** | **Shuffled** |
| 10 | 0.000000285 s | 0.000000155 s | 0.000000282 s | 0.000000203 s |
| 1000 | 0.000013640 s | 0.000007153 s | 0.000008175 s | 0.000006091 s |
| 10000 | 0.000132888 s | 0.000065965 s | 0.000079206 s | 0.000045531 s |
| 100000 | 0.000132888 s | 0.000065965 s | 0.000079206 s | 0.000045531 s |
| 1000000 | 0.006518392 s | 0.003235113 s | 0.000755829 s | 0.000856151 s |

Quick Sort Analysis

The most noticeable trend in quick sort’s data is that it outperforms merge sort across every type of array of every size. This is likely due to our implementation of quick sort, which used a middle value as the pivot point in the partition phase of quick sort. While the middle value is not necessarily the best value to use, it guarantees that we avoid quick sort’s worst-case O( running time. Additionally, it is possible that the middle value will be the best selection. For instance, in ordered and reverse ordered array, picking the middle value is ideal. This will lead to an O(N) running time instead of quick sort’s average-case running time of O(NlogN).

Like merge sort, quick sort’s worst performance occurred on the random array. The explanation for this is likely the same as Merge Sort’s explanation. With a randomized array, it’s possible that the algorithm ends up performing far more comparisons than it would in an ordered or reverse ordered array. This would also explain why the shuffled array performs slower than the ordered and reverse arrays. Whenever random data is used, we cannot guarantee minimal comparisons for the algorithm.

Like merge sort, quick sort has the potential for stack overflows. However, like merge sort, this issue is resolved by properly de-allocating memory from the stack.

Conclusion

This analysis clearly shows the benefits of using divide and conquer sorting algorithms instead of linear algorithms. Although each of the O( have the potential of performing quickly, their best-case scenarios are highly dependent on very specific sizes and types of arrays. Merge sort and quick sort, however, regularly perform in O(NlogN) time. Quick sort, specifically, has the potential of running in O(N) time, which is better than all the other algorithms discussed. Ultimately, this report suggests there is little reason to ever use a linear sorting algorithm when a divide and conquer algorithm is available.