Algorithmic Analysis of Quick Sort and Other Derivations

Robb Hill
Metropolitan State University Student

ABSTRACT

Algorithms all have a time complexity component. That component is typically dictated most by the order of magnitude of the algorithm. However other coefficients in the equation do have an influence on the time complexity. This is an experiment to try all combinations with the aim of proving the coefficients have a considerable, or substantial effect on the runtime of a quicksort program.

INTRODUCTION

Computers are tasked with operations such as sorting arrays, which like all operations in a computer follow an algorithm based on mathematic principals. Just like how not all values of infinity are equal, so too is true of algorithms. The efficiency of an algorithm is key to its appropriateness for a solution and is paramount to its selection.

Though the size of an algorithm under load is a concern in some cases, mainly the efficiency of an algorithm is measured in its time complexity. Poor efficiency of an algorithm causes unnecessary work. Unnecessary work causes execution growth in both space and time complexities. Minimizing superfluous actions makes an algorithm fast. A fast algorithm can seem like it may have a minimal effect on a set of data or a solution however, algorithms aid in the comprehension of scaling problems to appreciate the growth, linearly or exponentially in its complexities. After all, algorithms do the heavy lifting of large problems and are typically applied to massive data sets.

There are many different measured levels of efficiency for algorithms. Typically, we encounter four main types. There are, logarithmic, linear, linearithmic and quadratic. These assume the following forms:

log n	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
n log n	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on n × n matrices are standard examples.

Table 2.2 pg. 59 Anany Levitin: Introduction to the Design and Analysis of Algorithms

The main question we wish to explore in this paper is the possibility that mixing algorithm types have any gains to be measured. As a rule, we know that typically time complexity takes the highest order of the equation. We will experiment with various sorts to determine the differences in execution time.

METHOD

There are two main partitioning methods of quicksort, the Hoare partitioning scheme and the Lomuto partitioning scheme. The difference is subtle however Lomuto's scheme moves systematically from one end of the array to the other. In contrast, Hoare's scheme moves the pivots from both sides of the array simultaneously.

Next to consider is the placement of the pivot point within each partition for sorting. We will analyze the difference of Left side moving pivots, right side moving pivots or median value pivots. They operate as they sound, starting at the left, right or median value of the partitions within the array and moving across to the opposite side or outward from the median.

Lastly, we must look at having a minimum partition size before switching to insertion sort. Insertion sort is extremely fast on nearly sorted arrays. The closer the array is to sorted, the faster the algorithm can work. We must consider at what size of partition we wish to switch to insertion sort to maximize efficiency.

To approach this problem, I have written a program to test all combinations of methods with decreasing values for minimum partition size. We will use an array of 1,000,000 16-bit unsigned integers. That is that the sorts will be conducted on the same, randomly generated array of 1 million values, for the cartesian product of all options. For both Hoare and Lomuto schemes we will test each pivot of left, right and median, decreasing the minimum partition size for all with each iteration.

RESULTS

Sort Type	Pivot Type	Minimum Partition Size	Completion Time (in nanosecond)
Hoare	Left	10	30398500
Hoare	Right	10	11223300
Hoare	Median of 3	10	4387500
Lomuto	Left	10	8994000
Lomuto	Right	10	4669800
Lomuto	Median of 3	10	2118370

At this point I gave up. I ensured in the code that the array was the same unsorted array at the start of each sort algorithm. I could not figure out what was causing the same sort combinations, on the same starting array, to produce different results with every run. I spoke to you about this at length, and we were unable to determine the cause of the anomaly.

DISCUSSION

It is difficult to discuss the inconsistent results. Here is what we know; the time complexity of an algorithm is mathematically computed at the highest order of the mathematic representation. However, with that said there are still coefficients at work that, though minimally, effect the final values of time to complete the sort. That small margin is the entirety of this exercise: shaving off nano seconds with each permutation of sorts. The coefficient is not significant enough to affect the order of magnitude for the time complexity, but the results would be apparent in an environment that works statically, without interference.

My understanding is that the Lomuto method has lower coefficient effect on the time complexity due to the constant effect the Hoare method is compounded by because the work is run in parallel. Additionally, the pivot from left to right makes little to no difference and the median choice ads a complexity to the algorithm which is unnecessary and inefficient. Lastly, the insertion sort would require extensive testing in a working environment to determine the balance point of efficiency vs time complexity. Ultimately the balance point would have to be an average of many runs of the experiment because the value would likely be different depending on the spread of values, repeat values and the nature of the “shuffled” array. Without knowing how close an array is to being sorted there is no way of telling what point would be most efficient.