Jake Shapiro
Martin Glusker
April 12 2021

An Analysis of Quicksort

Intro

In this project, we examined the question of how to make quicksort most efficient and used this knowledge to create a better variation of quicksort. We evaluated the performance of variations of the quicksort algorithm on different input types, and then created a modified deterministic quicksort to improve performance. We analyzed six applications of quicksort by changing the algorithm by either setting the partition value deterministically or randomly, and by initializing the input array with either a partially sorted, mostly sorted, or unsorted array. Observing performance of quicksort on inputs that are sorted to various degrees can help identify the optimal quicksort method for a given input, and help us find an algorithm that is efficient regardless of the initialization of the original array.

Making quicksort more efficient is an important problem because small variations in an algorithm can have massive impacts on runtime. They can make theoretically solvable problems unsolvable in practice. Further, understanding what makes an algorithm work efficient or poor is important for deciding its optimal applications. For example, we sought to learn how we might change the way we sort if we have limited information about the characteristics of the input array.

Our goal for this project is to figure out which variations of quicksort perform well and why they outperform their counterparts. We then apply this knowledge to create a better performing version of quicksort.

Methods and Experimental Setup

We began by implementing a quicksorting algorithm based on the pseudo-code presented in class. We included function arguments that could specify whether the algorithm would be run randomly or deterministically. For random, we selected a random index between lo and hi, and for deterministic, we selected hi.

Arrays used for sorting were initialized using the code from Professor Gardner. They were int arrays and were either random, partially sorted, or mostly sorted.

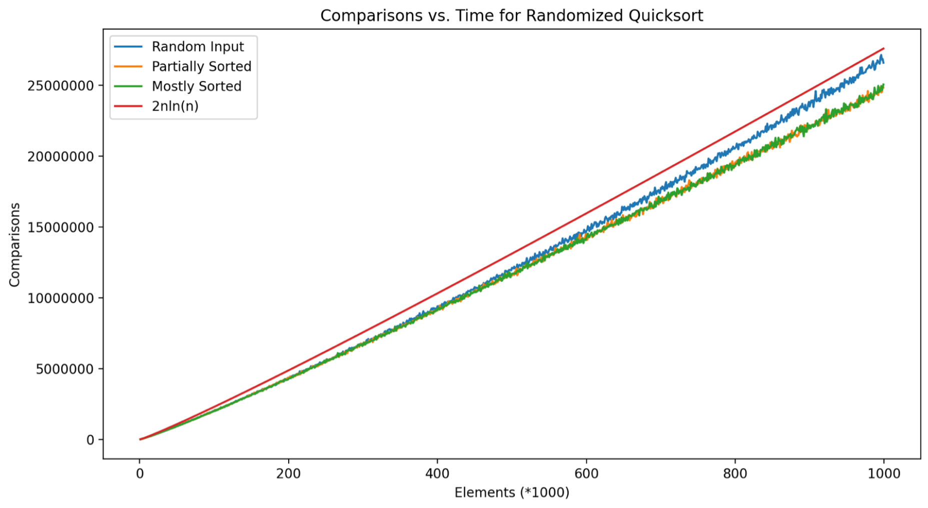
To measure runtime of random quicksort, we initialized 10 arrays at each multiple of 1000 between zero and one million. We then sorted each array of the ten arrays at each n and found the average number of comparisons. We did this process for random inputs, partially sorted inputs, and mostly sorted inputs. Data was written to a csv file to be processed using the pandas and matplotlib python libraries. We then graphed the data (elements on x-axis and comparisons on y) to compare the runtimes of random quicksort on each of the three input types. For deterministic quicksort, we used smaller arrays (up to n=200000) because the performance was bad enough that it caused stack overflows on larger values. Otherwise, the process was the same except for the difference in the sorting algorithm.

Next, we evaluated the variance in the number of comparisons. For each sorting method, on each type of input, we sorted 100 arrays of sizes 100,1000, 5000, 10000, 15000, and 20000. We then measured the variance of the sorting times among those arrays. We used python's pandas library to do the data processing and calculations.

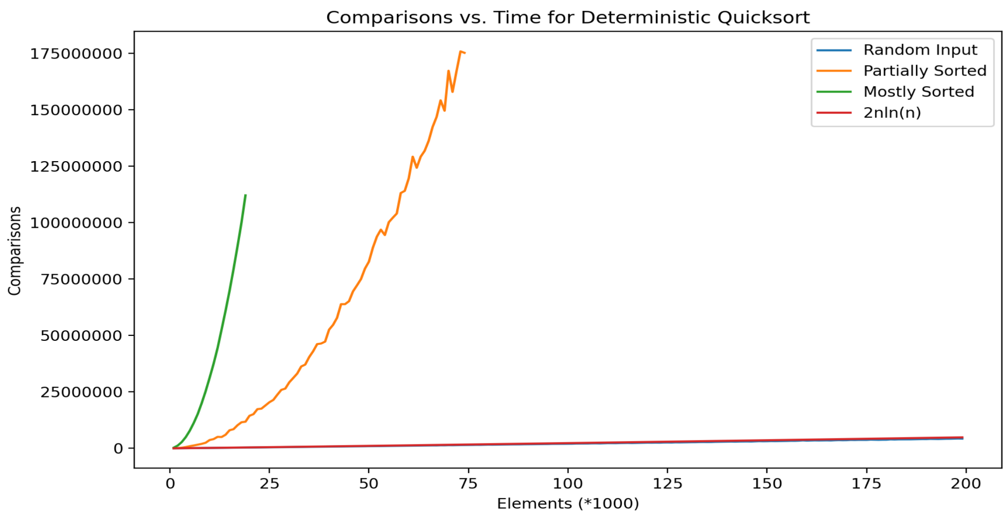
Finally we implemented our own version of quicksort. Our method selects r using the formula r=(hi+lo)/2. We then repeated the experiment to measure runtime from above, this time using our sorting method. We graphed the results to analyze runtime.

Results and Discussion

We found that randomized quicksort on all three types of input performed similarly to the theoretical runtime of quicksort of nlg(n):



Deterministic quicksort on random input performed similarly to the theoretical runtime, but on partially sorted inputs and mostly sorted inputs, it performed far worse (The random input line is under the theoretical line):

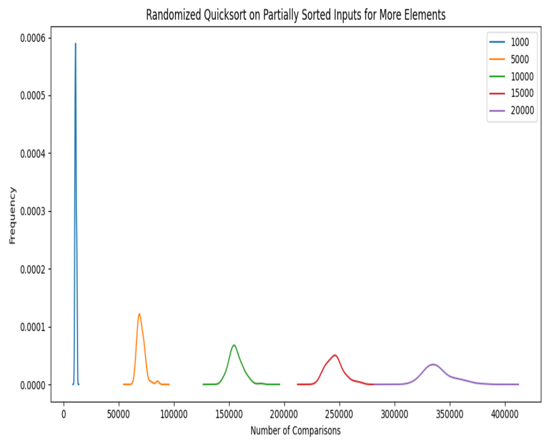


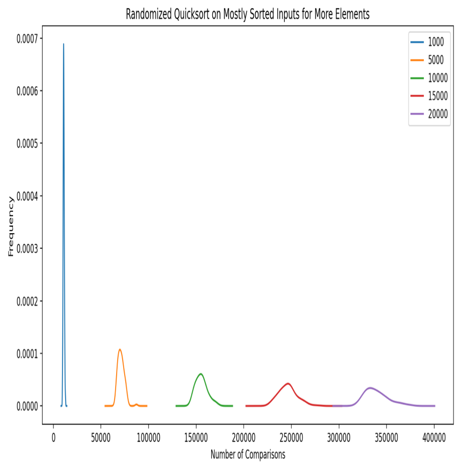
Variance in number of comparisons was similar across the three types of input for the randomized quicksort as well as random input for deterministic quicksort. We measured their squared coefficients of variance and reported them below for sorting on 15,000 elements, though it was equally similar for all numbers of elements.

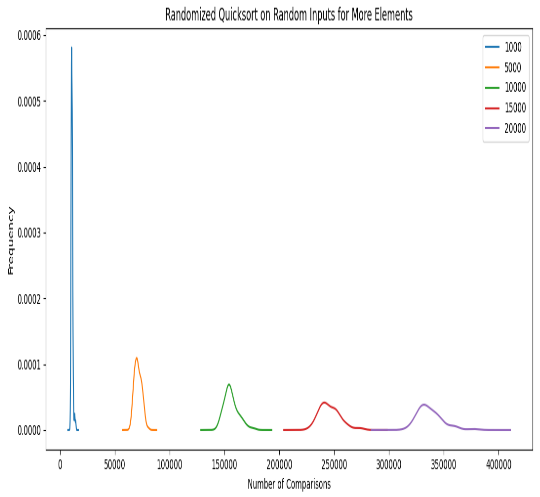
Squared Coefficient of Variance for 15,000 elements (normalized values in parentheses and similar results highlighted):

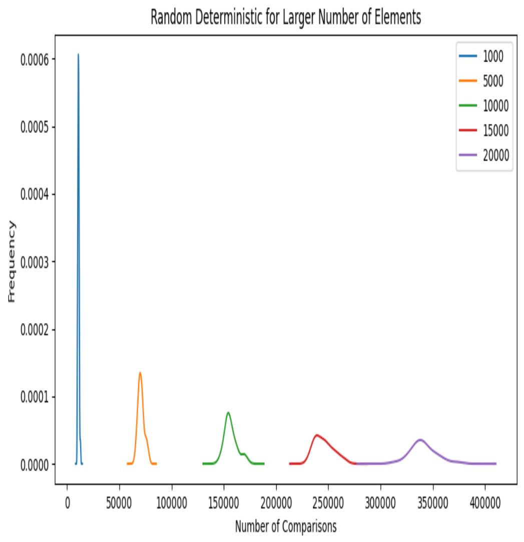
	Random	Partially Sorted	Mostly Sorted
Random	0.00146 (0.0495)	0.00106 (0.0359)	0.00152 (0.0515)
Deterministic	0.00133 (0.0451)	0.02936 (.99) (high)	0.00011 (.003) (low)

The kernel density function graphs of the numbers of comparisons below display the variance. The graphs are nearly the same for any sorting that involves some element of randomness, and there is more variance on deterministic/partial and less variance on deterministic/mostly.

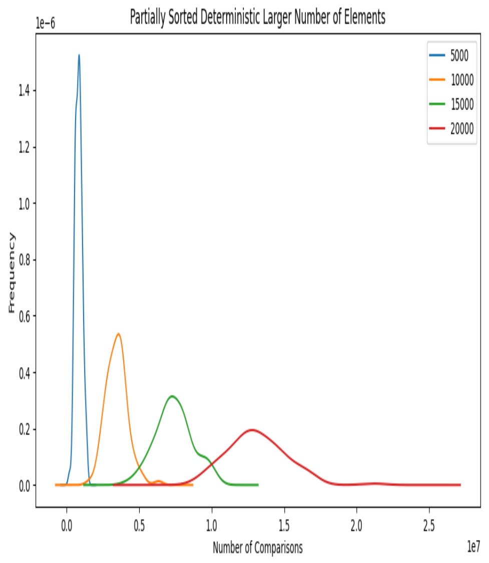


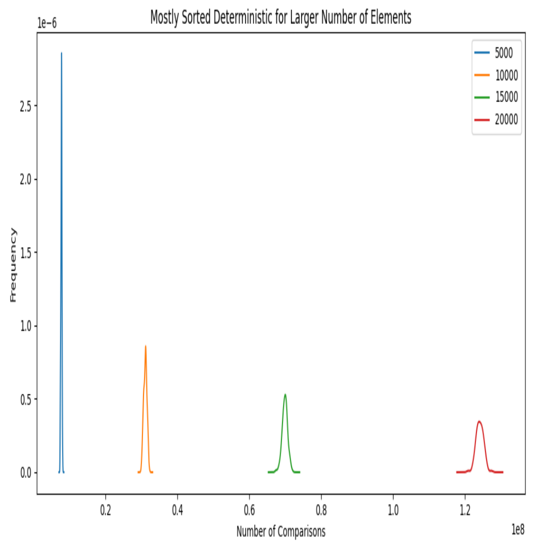






These four graphs are essentially the same.





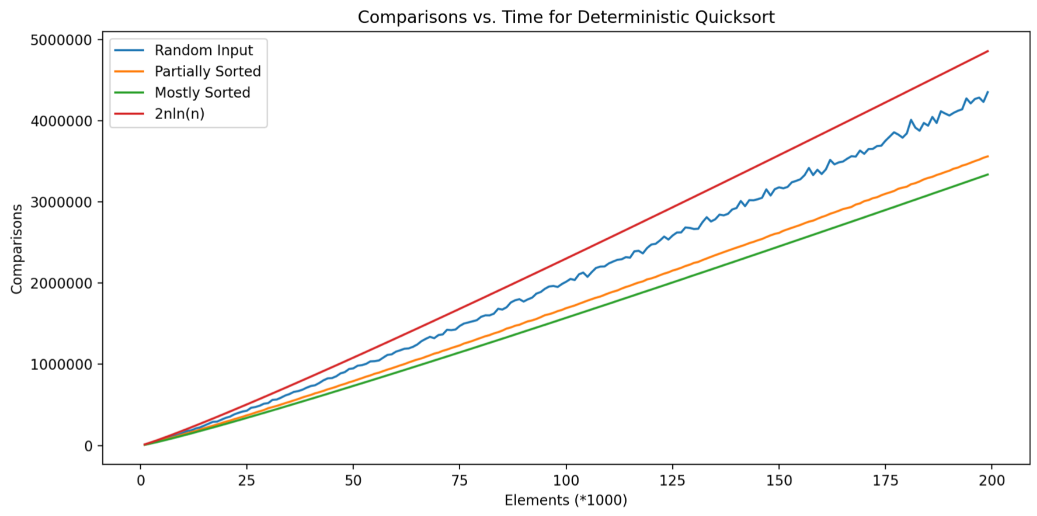
The four experiments that involve a random component show very similar variance, while deterministic quicksort on mostly sorted and partially sorted inputs had the least and most variance, respectively. Closer inspection reveals that the partially sorted input could yield numbers of comparisons as good as random sorting or as bad as deterministic sorting on mostly sorted input, leading to high levels of variance. Deterministically sorting mostly sorted inputs consistently had numbers of comparisons far above the other sorting methods, which led to its low variance.

From these data, we can conclude that quicksort performs the best when there is at least one element of randomness, either in the algorithm itself or the input. It makes sense that these variations of quicksort display similar results because they each select the partition element randomly, making them functionally identical.

Quicksort works best when the partition element is close to the median of the elements in the array. If the input to the sorting algorithm is random, it is not possible to predict where in the array the median will be, and no element can be expected to have any specific value relative to the rest. When the original data are mostly or partially sorted, as with our experiment, we can expect on average that the element in hi is larger than the other elements. Our deterministic algorithm is expected to pick values that are larger than most elements in the array, rather than elements in the middle, which leads to a significantly higher runtime. When the highs and the lows of quicksort are uneven in size, we lose the benefits of dividing and conquering, and runtime increases towards polynomial time.

Randomization improves the runtime of quicksort because it becomes less likely to choose the higher elements as partition elements and is thus more likely to find an efficient partition. In improving quicksort, we sought to maximize efficiency by actively trying to choose a good partition element, rather than probabilistically avoiding bad ones. In a randomized input to quicksort, the performance nears the theoretical limits, but on partially or mostly sorted input, the performance is much worse. We hypothesized that we could improve the runtime of a deterministic quicksort on these inputs by selecting our partition index to be the midpoint of the array. Because these arrays are partially sorted, the hi index is more likely to be a higher value, and the middle index is more likely to be a middle value, which is optimal for quicksort. We thus created a new quicksort where r is selected as (hi+lo)/2, so a[r], the partition element is more likely to be a middle value on sorted data and still a random value on random data. We hypothesized this would improve runtime for partially and mostly sorted data, while maintaining the same efficiency with randomly sorted data.

Our data supported our hypothesis and the new algorithm beat the theoretical runtime on random, partially sorted, and mostly sorted inputs:



Given these results, it is clear that selecting the midpoint of the array is more likely to select a median value for partitioning the mostly or partially sorted data, while maintaining a random selection for the partition value on random data. Thus we achieve the same magnitude of performance of randomized quicksort for all types of inputs, improving upon the previous iteration of quicksort.

Our quicksort algorithm is good because it meets the performance benchmarks of fully randomized quicksort but is deterministic, which, in some situations, is valuable in an algorithm because it allows repeatability and better analysis of the results.

Conclusion

It is clear that the efficiency of quicksort depends on the algorithm's ability to select a good partition element, ideally one whose value is near the median value of the array. When input to the function is random, it is impossible to predict where the median value is in the array, and thus any partition element is equally likely to be good or bad. When input arrays are not random, but partially sorted, we can create strategies to minimize the probability of picking a bad partition element. We have shown that randomly selecting our partition element makes us less likely to select a value near one of the extremes as our partition, and randomized quicksort is efficient. We further showed that selecting an element near the middle of the array is likely to result in the partition value being near the median when the data are partially sorted. Randomization can allow us to avoid making bad choices when performing quicksort, though there are certain inputs where we can perform better by trying to predict the location of the median data element.