COSC223 Probability and Computing
Prof. Gardner

Shakhzod Ali-zade
Daniel Njoo
April 12, 2021

Project 1: An Analysis of the Quicksort Algorithm

Introduction

The goal of this project is to analyze the difference in performance between the Deterministic and Randomized versions of the Quicksort algorithm, evaluate which is better, and look to enhance it if possible. Sorting, the problem of putting elements in a list in a certain order, is one of the most common and important problems in computer science as it is commonly used within other algorithms. So improving sorting algorithms is incredibly beneficial to the field at large.

Quicksort is one of the faster known sorting algorithms, with a best and average runtime of $O(n \ \log n)$, and is a divide-and-conquer algorithm that involves selecting a “pivot” element from the list of elements to be sorted and then recursively sorting the two partitioned sub-lists. The difference between the Deterministic and Randomized versions of the algorithm lies in how the pivot is chosen. For the Deterministic version, the pivot is chosen from either the first, middle, last, or median element, while the randomized version picks the pivot element uniformly at random.

We tested the performance, measured as the number of swaps made, of the two versions of Quicksort on input lists of various lengths and three types: unsorted, partially sorted, and mostly sorted. We then averaged our results over multiple runs (n=20) and also calculated the variance and normalized variance of the runtimes. Comparing the two versions’ performances over different input lengths shows us how they fare compared to the theoretical runtime bounds and comparing them over inputs that were sorted to different degrees allows us to see the difference in performance that might be caused by the different input types. We also implemented a different partition function that uses the median element of the list as the pivot as a potential improvement.

Methods and Experimental Setup

Our Quicksort was based on the pseudocode presented in class, and counts the number of comparisons made during the sorting process. The input arrays were generated using the code provided by to us in `Generate.java`, and consist of one of three types:

1) random
2) partially sorted
3) mostly sorted

Example arrays of size 10 are shown below that depict the varying degrees of sortedness, note that the size of the number is irrelevant here, only the order or lack of order is pertinent to the sorting problem:

Type	**Array**
Random	[27902, 52797, 40629, 99114, 53482, 24581, 50409, 24560, 52219, 61034]
Partial	[20, 107, 97, 165, 257, 285, 319, 403, 444, 471]
Mostly	[45, 73, 126, 185, 249, 367, 400, 361, 456, 413]

In actuality, the program creates arrays of each of the three above types in different sizes, either of the 3 sizes {1000, 10000, 20000} for Deterministic Quicksort -- higher input sizes resulted in stack overflow errors -- and 11 sizes for Randomized Quicksort: {1000, 10000, 20000, 50000, 100000, 300000, 400000, 800000, 1600000, 3200000, 10000000} and then runs the two versions of the algorithm 20 times and averages the runtimes. Using this average we then calculated the variance and normalized variance of the runtimes defined as:

$$Var = \frac{\sum_{i=1}^{m} (x_i - \bar{x})^2}{m-1}$$

where m=20, x_i is the ith runtime and $\bar{x}$ is the average runtime across the m runs

and

$$C^2_X = \frac{Var(X)}{E[X]^2}$$

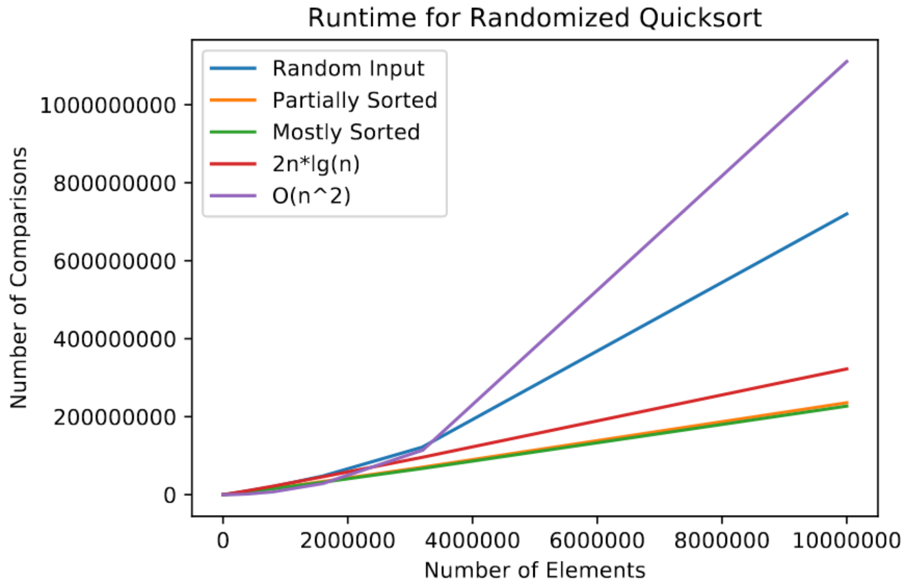
respectively. We also implemented a modified partition function where we used the median element instead of the highest element, to see if it would improve the Quicksort algorithm's run time.

So in total we ran the following three experiments, where the independent variable was the 3 or 11 different sizes of array and the dependent variable was either the runtime or variance:

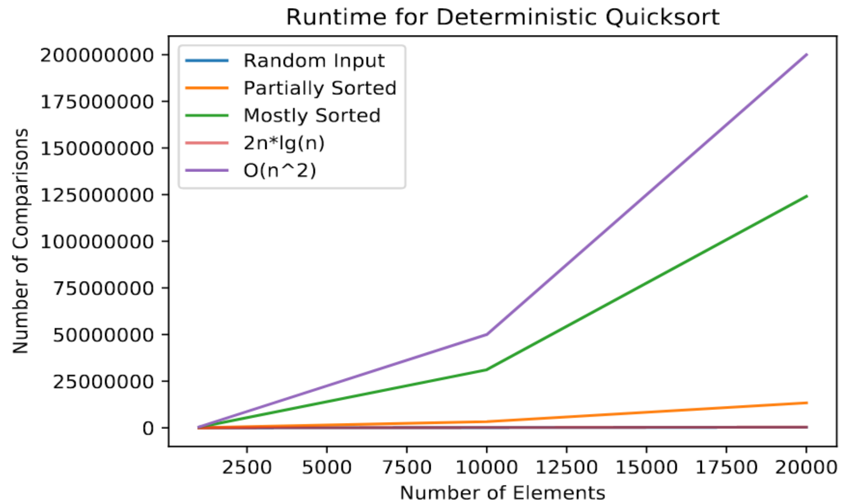
Experiment	**Permutations**
Input's degree of sortedness	3 degrees: random, partial, mostly sorted
Deterministic vs random	2 types: deterministic or random
Median partition function	2 types: highest element or median

Results and Discussion

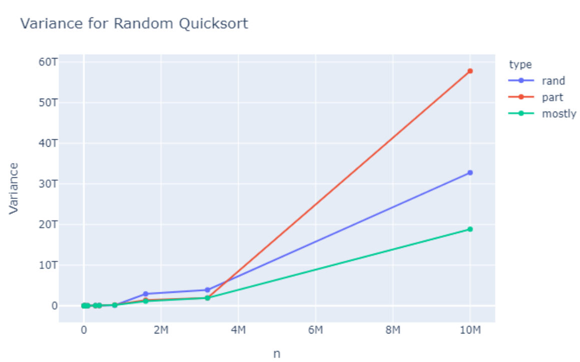
We found that Randomized Quicksort performed optimally, close to $O(n\ lg(n))$ (in red) bound on partially or mostly sorted inputs, but performed worse on randomized input and closer to the upper bound of O(n^2) (in purple). However, since we have O(n^2) = $\frac{n^2}{900000}$ for Randomized Quicksort, its performance is still optimal.

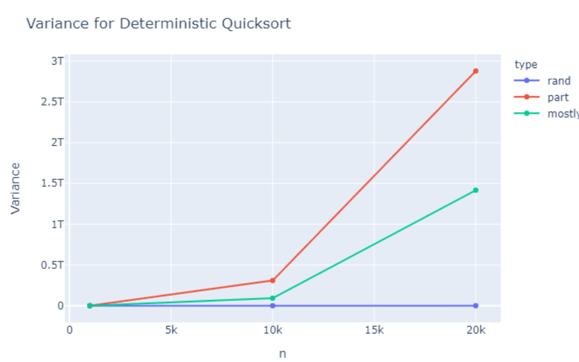


Overall, however, Deterministic Quicksort performed worse than the randomized version. Sorting a mostly sorted array of the size larger than 20000 caused memory overflow in our IDE, and was the reason why we used arrays of size only up to n = 20000. Runtime on mostly sorted input was almost quadratic time (O(n^2) = n*n (in purple)), and sorting partially sorted arrays was lower bounded by $O(n\, lg(n))$(in red). Sorting randomized input, however, was consistent with the $2 * n\, lg(n)$ runtime bound we expected.

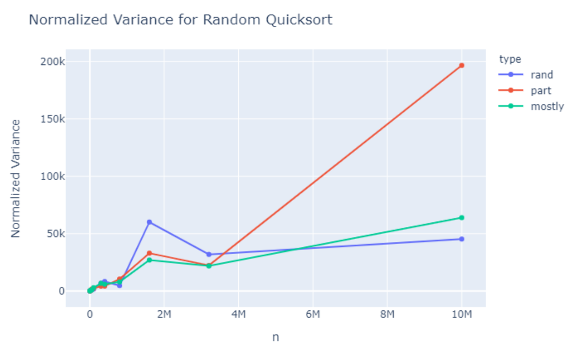


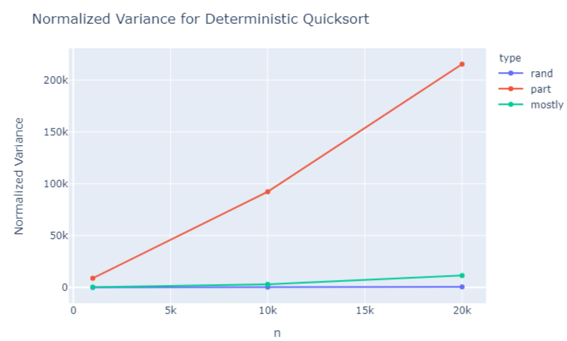
The variances for both Quicksorts are shown below and though we expected randomized input to result in higher variances, we found that partially sorted inputs resulted in the highest variances. Additionally, though the y-axes are scaled differently because we ran deterministic Quicksort over a smaller range of input lengths, we found that randomized Quicksort exhibited generally higher variance as seems intuitive.





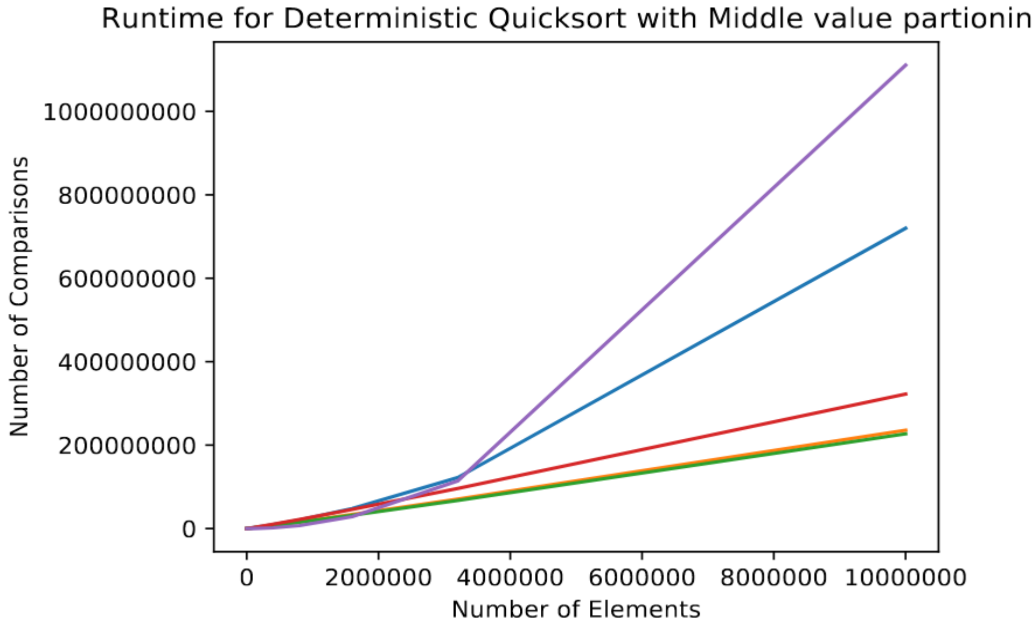
The variance values scale into the trillions thanks to the size of the inputs, so normalized variances were also calculated and are shown below. Again we see the same results: higher variance for partially sorted inputs over others (not intuitive) and Randomized over Deterministic (intuitive).





Therefore, we identified the overall effect of pivot randomization as resulting in a faster runtime, and input randomization as resulting in a slower runtime. This has to do with the probability of choosing a bad pivot, especially for Deterministic Quicksort. Therefore, using Randomized Quicksort is generally preferred. However, we found that randomized Quicksort vastly underperforms on randomized input while Deterministic Quicksort struggles on mostly sorted input.

We hypothesized that choosing the last (`hi`) element of the given range led Deterministic Quicksort to choose extreme values as the pivots. We further hypothesized that it will perform better if we choose the median element in the given range as the partitioning element, because the median is not located on the ends of the given range and is thus less likely to take an extreme value. To test this assumption, we made the partitioning element to be equal to the middle value in the given range (i.e. ind = (hi+lo)/2). The results of the experiment shown below supported our hypothesis, and the performance of the quicksort has improved. The number of comparisons decreased by a factor of thousands. Furthermore, if before we could not sort arrays of the size larger than 20,000, now we were able to sort up to arrays that had 10,000,000 elements. Therefore, the findings strongly supported the hypothesis that choosing the middle element in the given range improves the performance of the Quicksort.



Conclusion

This analysis of the Quicksort revealed that Randomized Quicksort generally performs better than the Deterministic Quicksort because Randomized Quicksort has a lower chance of selecting a bad pivot. However, Randomized Quicksort underperforms on randomized input while Deterministic Quicksort underperforms on mostly sorted input. Overall, pivot selection is the critical piece of the algorithm as choosing bad pivots can considerably lower performance, evidenced by the worst-case time complexity of $O(n^2)$when compared with the best and average-case of $O(n\ lg(n))$. One way to mitigate the possibility of choosing bad pivots is to either use randomized Quicksort or seek to use a partitioning function that selects the median value. We showed that using this partitioning function improves the performance of the deterministic quicksort. In further analyses one might hypothesize that using another sorting algorithm, like Insertion Sort, in combination with Quicksort might improve the performance of the Quicksort even further. Because Quicksort in general performs worse on partially sorted and mostly sorted arrays than Insertion sort, combining both of them could lead to a better result.