Project 1: Quicksort

Investigating Randomization in Algorithms through Quicksort

Hope Tsai Sara Kao

April 12, 2021

Introduction

Randomization is an inevitable occurrence in the natural world, yet an elusive idea in a digital world that follows predetermined, `if-then` paths. However, in this age of innovation, randomization is becoming increasingly important. Thus, as it becomes widely used in new algorithms, it is crucial that computer scientists understand how it factors into established ones. One such example is Quicksort, which sorts arrays. Partitioning a given array into two sections, the "bigs" and the "smalls," Quicksort recursively calls itself on each side of the partition until the entire array is sorted. In this paper, we study deterministic and randomized implementations of the Quicksort algorithm, which differ only in their partition selection methods. The former always picks the element at position `hi`, the highest index of the subset of the array, to be the partition. By contrast, the latter randomly chooses the partition element, where each index in the array is equally likely to be chosen.

We compare the average runtimes of the deterministic and randomized implementations for arrays varying in size and level of initial sorting (completely randomized, partially sorted, mostly sorted). The average runtimes are measured by the number of comparisons between array elements. We also compare the variances of the runtimes, between the two implementations, for arrays of size 10,000. We observe that, relative to the randomized implementation, the deterministic implementation generates consistently higher average runtimes, with lower variance between runtimes. Using these observations, we modify the Quicksort algorithm, in an effort to achieve both the shorter average runtimes of the randomized implementation and the lower variance of the deterministic. We then repeat our original experiments on the modified algorithm to determine whether or not we achieved our goal.

We use these analyses to better understand the benefits and shortcomings of randomization in one application: Quicksort. We aim to assess the differences between its randomized and deterministic implementations and to propose potential improvements to the partition selection method. Ultimately, our goal is to visualize and realize the effects of randomization on Quicksort. Thus, we hope to better understand the role that randomization can play in simplifying and enhancing algorithms. By doing so, we broach the question, "Can one innovate and improve upon an established algorithm?"

Comparing Randomized and Deterministic Implementations of Quicksort

Experimental Set-up

The initial experiments were divided into three categories, based on the array's initial level of sorting. Generated by the given Generate class, the arrays we studied could be 1) completely randomized, with equally likely inputs and orderings, 2) partially sorted, or 3) mostly sorted. We tested all array categories with both randomized and deterministic Quicksort, analyzing the average runtimes as the size of the array varied. We also analyzed the variances of runtimes for arrays of size 10,000. In the terminal, the program takes in user input to select the appropriate array category, version of Quicksort, and type of experiment (average runtimes or variance).

First, we analyzed the average runtimes for completely randomized arrays. For both randomized and deterministic quicksort, the smallest array size analyzed was 10,000 elements, and the largest, 1,000,000 elements; the size was incremented by 90,000 elements until it reached the maximum size. (The program will prompt the user for these values.) For each array size, 100 arrays were generated and sorted using the specified implementation of Quicksort. Afterwards, the average was found and stored in an array of average runtimes, which was printed to the terminal at the conclusion of the experiment.

Then, we focused solely on completely randomized arrays of size 10,000, and ran 100 tests of randomized quicksort. Each test's runtime was stored in an array. The program printed the array, and calculated and printed the following: the average runtime, the variance, and the squared coefficient of variance. We repeated the process with deterministic quicksort.

Second, we analyzed the average runtimes for partially sorted arrays. For randomized quicksort, the smallest array size analyzed was 10,000 elements, and the largest, 1,000,000 elements; the size was incremented by 90,000 elements until it reached the maximum size. As before, for each array size, 100 arrays were generated and sorted using the chosen implementation of Quicksort. However, for deterministic quicksort, the aforementioned inputs resulted in a StackOverflowError. Thus, we decreased the maximum size by one order of magnitude. In the end, the smallest size analyzed for deterministic quicksort was 10,000 elements, and the largest, 100,000; the size was incremented by 10,000 elements. The average runtime for each array size was stored in an array, which was printed to the terminal at the end of the experiment.

For both implementations, we ran 100 tests (each) on partially sorted arrays of size 10,000. The program printed the same analyses as before.

Third, we analyzed the average runtimes for mostly sorted arrays. For randomized quicksort, the smallest array size analyzed was 10,000 elements, and the largest, 1,000,000 elements; the size was incremented by 90,000 elements until it reached the maximum size. Averages were calculated as previously discussed. For deterministic quicksort, these inputs resulted in a StackOverflowError. Thus, we decreased the numbers by two orders of magnitude. The smallest size was 100 elements, and the largest, 10,000 elements; the size was incremented by 100 elements until it reached the maximum sizes.

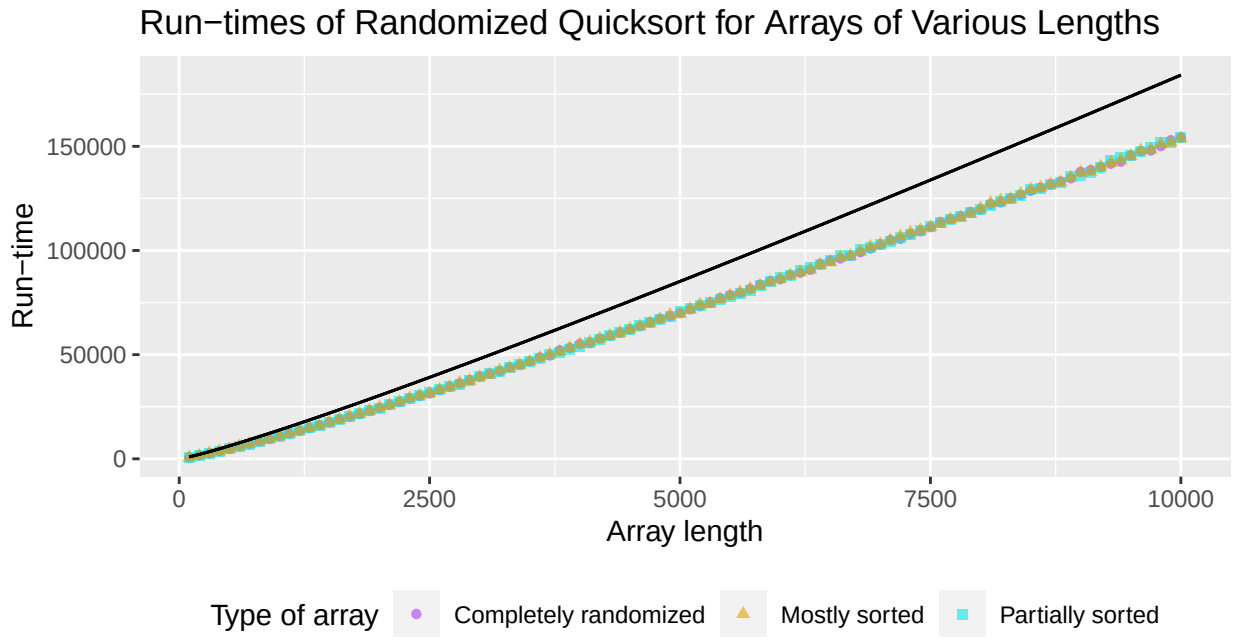
For both implementations of quicksort, we ran 100 tests on mostly sorted arrays of size 10,000. The program printed the same analyses as before.

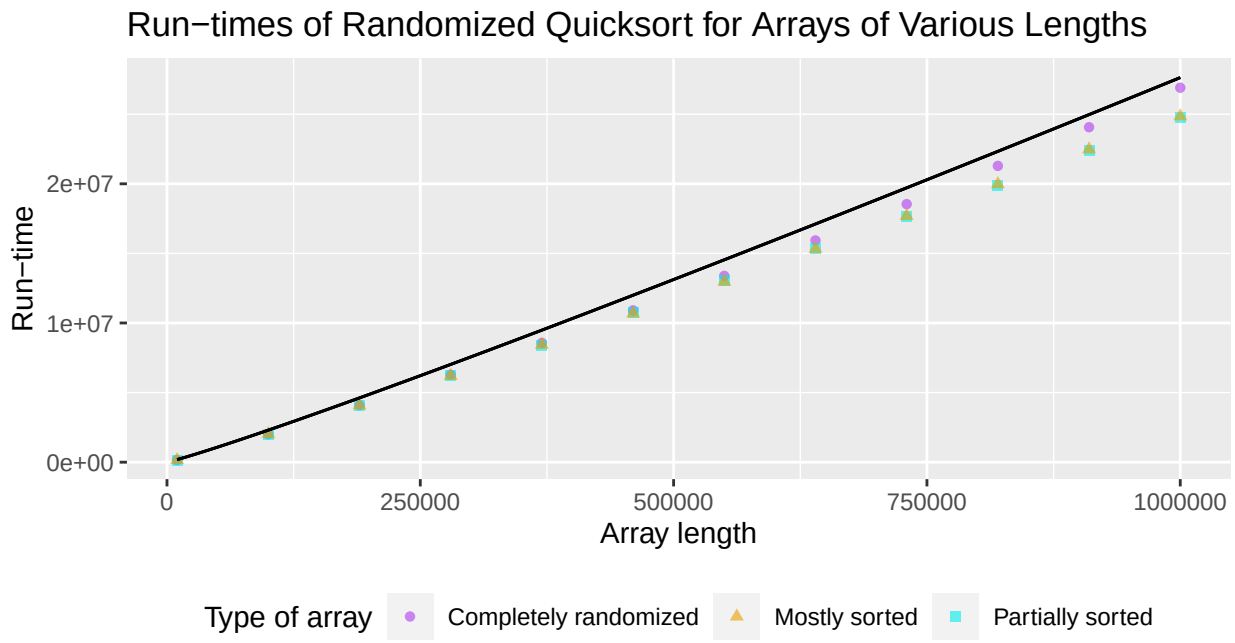
Results and Discussion

First, focusing on the algorithm's expected performance, we examined the differences between randomization over the input and over the algorithm's behavior. For each combination of the quicksort implementation, the type of array, and the array length, we calculated the average runtimes

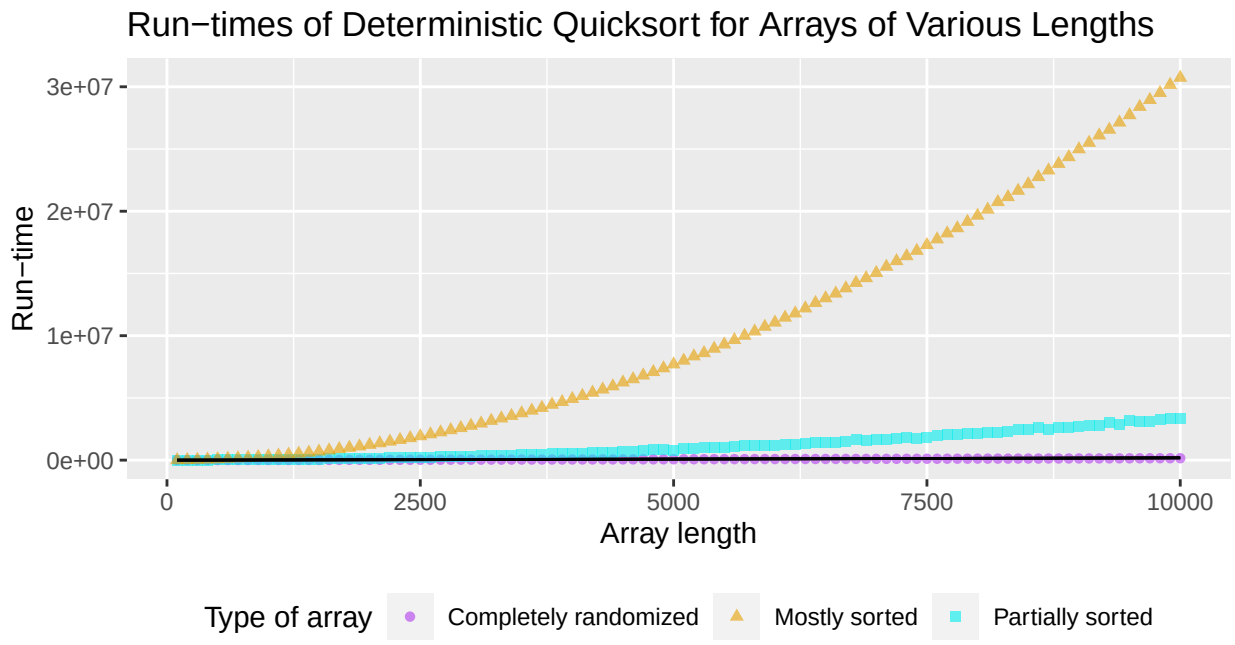
from numerous trials. We plotted the results using R, and included the randomization implementation's upper-bound runtime (in black) for context.

Analyzing Expected Run-time for Arrays of Varying Length

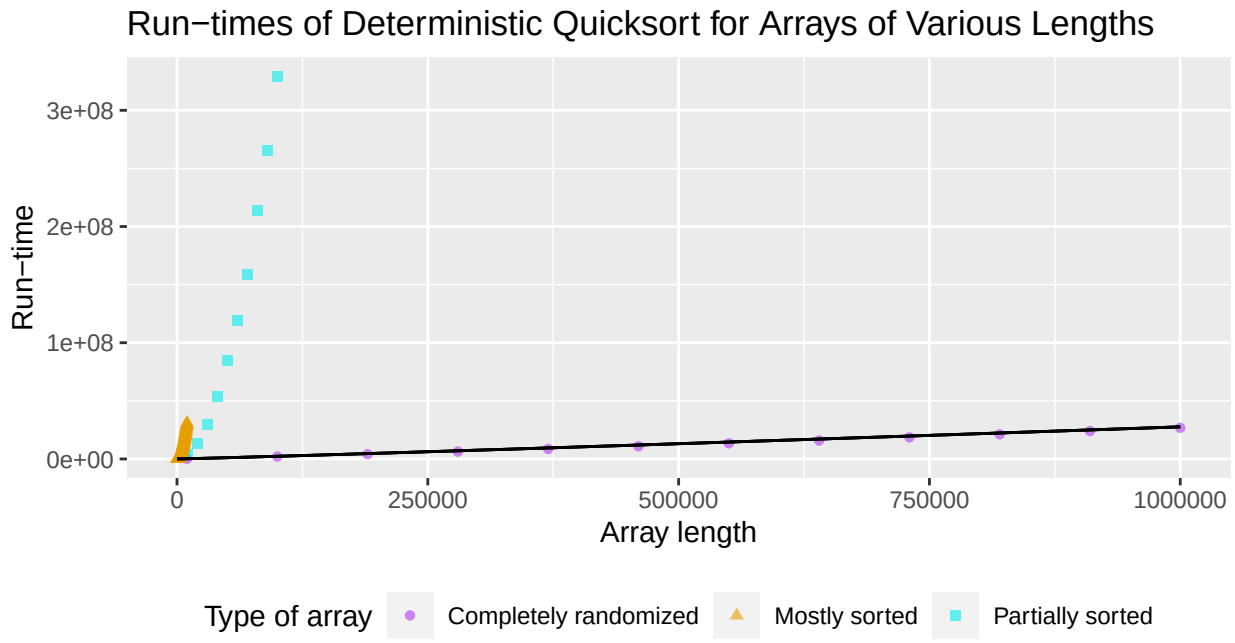





```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```




```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```



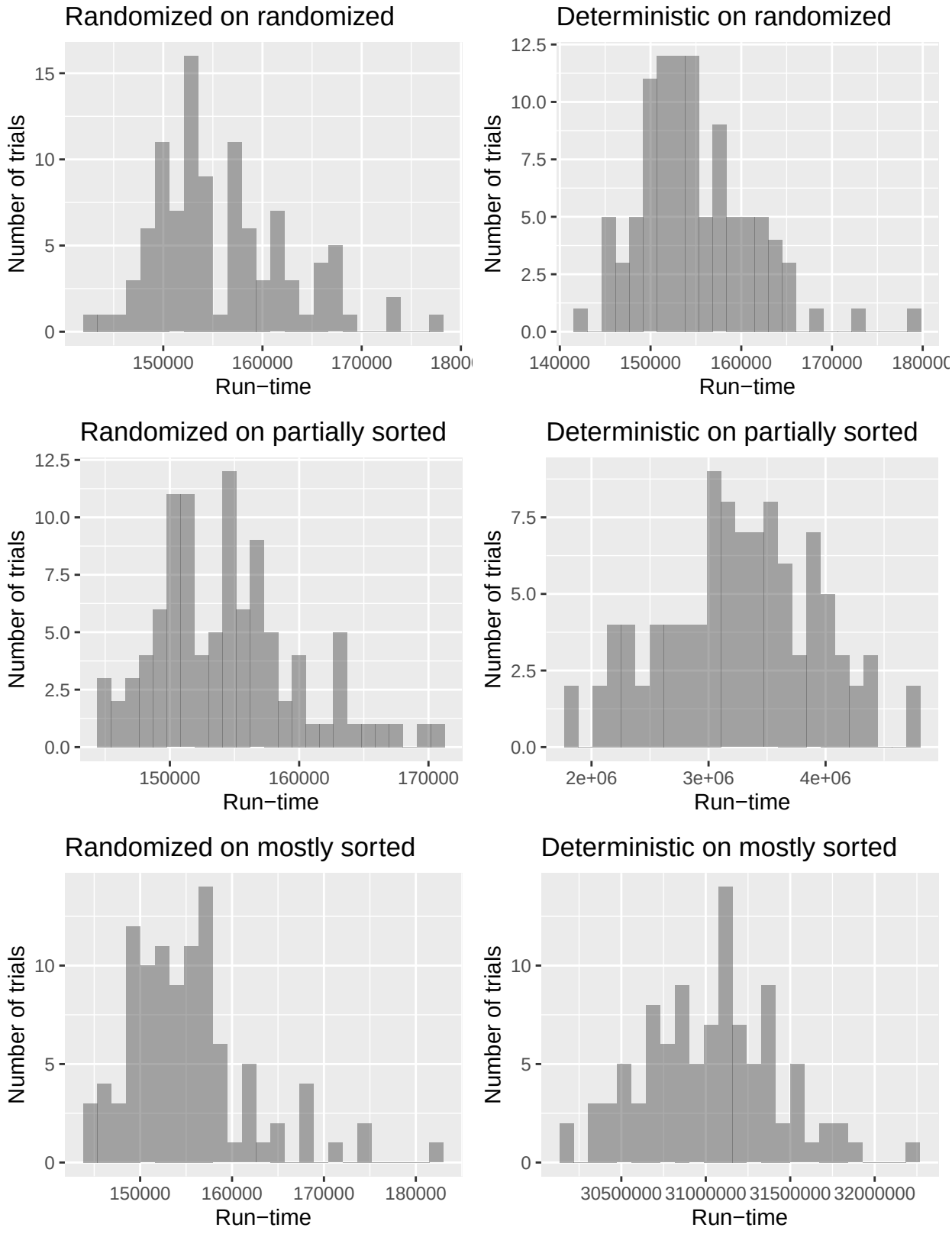
Though the graphs are on different scales, it is clear that randomization over the algorithm's behavior led to a comparably better runtime, relative to the deterministic implementation, across all input types; this may be attributed to the fact that the randomized algorithm is as likely to choose a poor partition element as it is to choose a good one, whereas the deterministic algorithm tends to choose poor partition elements, especially with partially and mostly sorted arrays. Moreover, randomization over the input largely did not counteract the benefits of randomization over the algorithm's behavior. The only exception was the completely randomized array input, which

had noticeably higher average runtimes for larger array sizes. However, across all array types and lengths, the average runtimes for the randomized implementation lay below the upper-bound runtime, and at or below the deterministic implementation's runtime.

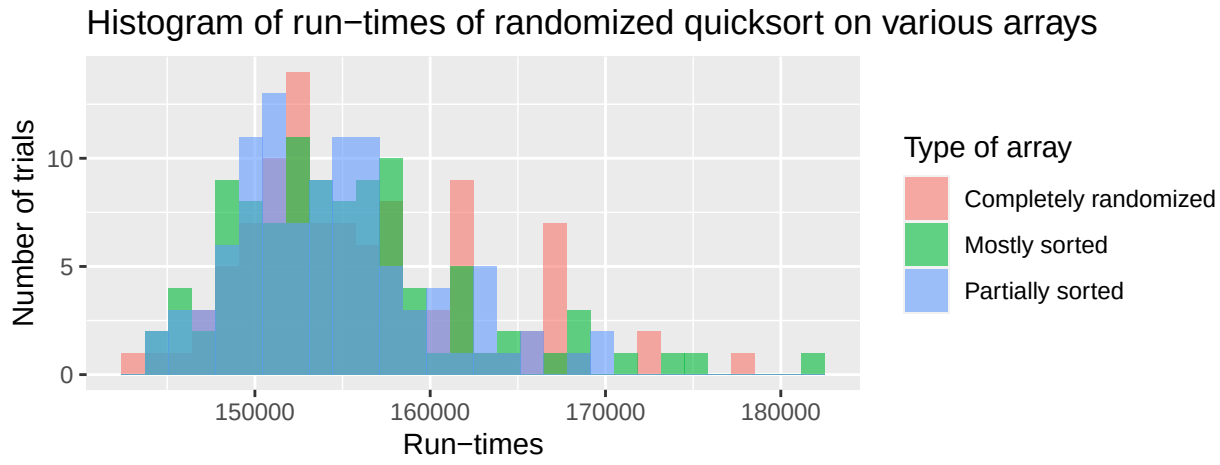
On the other hand, randomization over the input did influence the average runtime of the deterministic algorithm. In particular, those of completely randomized arrays mimicked that of the randomized implementation. Due to the unsorted nature of the input, even though deterministic quicksort always chose the element at position `hi`, it yielded random partition elements, thus producing behavior identical to its randomized counterpart. On the other hand, for any array length, the average runtimes of the partially and mostly sorted arrays were significantly higher than those of the randomized implementation. For these inputs, the element at position `hi` was more likely to be a higher value. Partitioning by such an element yielded unbalanced subsets of the array, which then increased the number of recursive calls necessary to sort the array. While both the randomized and deterministic implementations can choose poor partition choices, and thus create unbalanced subsets, the likelihood of choosing a relatively poor partition is much higher for these cases. As such, the average runtimes grew; for the program to run to completion, we had to adjust the sizes of the arrays analyzed.

By contrast, the average runtimes of the randomized implementation did not diverge significantly by level of sorting, and were consistently lower than those for the deterministic implementation. Consequently, we were able to run the randomized implementation on the same array sizes for all three inputs. Thus, we observe that the "divide and conquer" strategy of Quicksort was more-or-less wasted on partially and mostly sorted arrays using this deterministic implementation, and we conclude that, in general, it is more efficient to use the randomized implementation.

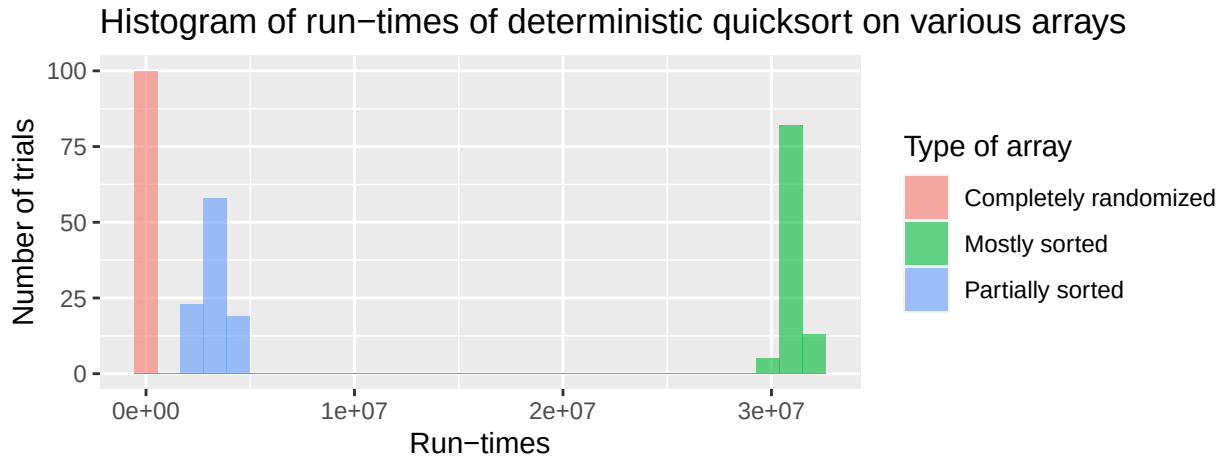
Analyzing the Spread of Runtimes for Arrays of a Given Size




```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Key Statistics of the Randomized implementation

	Average	Variance	Squared coefficient of variance
Randomized	154979.99	4.452835659585858E7	0.0018538961813511306
Partially sorted	155462.37	4.3080976235454544E7	0.0017825222834701776
Mostly sorted	155192.67	4.120050480919192E7	0.0017106459255114444

Key Statistics of the Deterministic implementation

	Average	Variance	Squared coefficient of variance
Randomized	154766.9	3.0571515545454554E7	0.00127632362
Partially sorted	3253085.15	4.2759771241964386E11	0.04040587452173949
Mostly sorted	3.111227964E7	1.6195375913681863E11	1.673121059447673E-4

Missing from our analysis was an understanding of the extent to which individual runtimes might vary from the expected performance. Thus, in addition to expected performance, our experiments also focused on the variance of runtimes for arrays of a given size. In repeated Quicksort trials on arrays of size 10,000, we observe that the variance of randomized quicksort was higher than that of deterministic quicksort for completely randomized arrays. At first, this result suggests that while the former implementation performs more quickly in expectation, the latter performs more consistently between tests.

However, it must be noted that the variance is sensitive to the mean. If the average runtime is higher, then the variance will also be higher. To adjust for this characteristic, we normalized the variance by analyzing the squared coefficient of variance. When we compare this new value for both implementations, we discover that the squared coefficient of variance is still higher for randomized Quicksort.

For partially sorted arrays, the results surprised us. In particular, the variance for the deterministic implementation was significantly higher than that for the randomized implementation; the former's variance was in the billions, whereas the latter's was in the millions. Even after we normalized the variances, we discovered the same result as before: the squared coefficient of variance for the deterministic implementation was higher than the value for the randomized implementation.

For mostly sorted arrays, the results surprised us in the same way: the variance for the deterministic implementation was significantly higher. However, the normalized variance, the squared coefficient of variance, was lower for the deterministic implementation and higher for the randomized. As such, the result of these tests can be grouped with that of the completely randomized arrays. The only instance in which the normalized variance was higher for the deterministic implementation occurred in the tests for partially sorted arrays.

We originally interpreted the relatively higher variances for runtimes of the randomized implementation relative to the deterministic implementation as follows: In randomly choosing the partition, the randomized algorithm could choose any element in the array, which may or may not be close to the actual median element. Thus, the runtime will vary depending on the quality of the partition choices, leading to both low and high runtimes, which in turn yield a high variance of runtimes. By contrast, especially for mostly and partially sorted arrays, the deterministic implementation almost always chose a poor partition, leading to consistently high runtimes and low variances between runtimes.

However, even after focusing on the squared coefficient of variance, we discovered that our

interpretation may not hold true for partially sorted arrays. Instead, perhaps, there are two groups in this input: "more sorted" partially sorted arrays and "more random" partially sorted arrays. Once 100 arrays have been generated, they may belong to either group. During deterministic Quicksort, the number of poor or good partition elements may depend on the array's level of sorting. (If the array is "more sorted," then the algorithm will more likely choose a poor partition, thus causing unbalanced subsets and leading to higher runtimes.) If the two groups occur in relatively equal numbers, then once 100 arrays have been generated and sorted, the variance may have risen significantly. However, we are not sure whether or not this interpretation is correct.

Summary

Through our initial experiments, the potential advantages of randomization are evident. The algorithm was not significantly more difficult to implement, and in terms of average runtimes, it matched (and often outperformed) the deterministic algorithm. Whereas the deterministic implementation faltered with partially and mostly sorted arrays, the expected performance of the randomized implementation on partially and mostly sorted arrays was actually better than its performance on completely randomized arrays. As such, it proved to be more generalizable: it could run at the same level of efficiency on arrays with various levels of initial sorting. Ultimately, we observed that randomization over the algorithm's behavior can lend efficiency, simplicity, and generalizability to algorithms, which in turn makes algorithms easier to utilize and analyze.

However, as noted before, randomization over the algorithm's behavior had one shortcoming: higher variances between runtimes for the majority of the inputs. We now begin another round of experiments in an attempt to alleviate this shortcoming. As the shortcoming arises due to the fundamental characteristic of randomization, we attempt to integrate deterministic characteristics into our algorithm to decrease the variance of the runtimes.

Modifying Quicksort

Experimental set-up

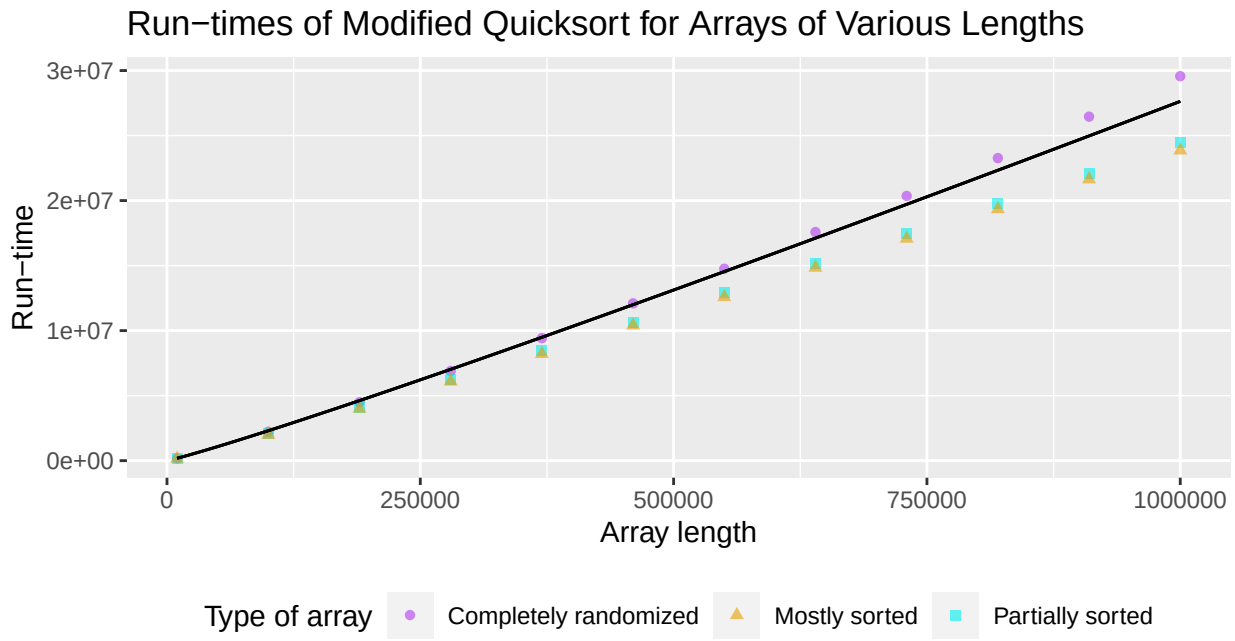
After our initial experiments, we hypothesized that we could improve the quicksort algorithm by utilizing elements from both implementations. We sought to take advantage of the lower expected runtime of the randomized implementation, while retaining the lower variance of runtimes from the deterministic implementation. We combine the key characteristics of both implementations. In the modified algorithm, all aspects of the Quicksort remain as previously specified, but we now choose our partition as follows: The program randomly chooses three indices, stores their elements in an array, sorts the array with bubble sort, and uses the median element as the partition element. This new partition-choosing method involves randomization—-in randomly choosing the three indices—-and determinism—-in comparing the elements at the three indices and always choosing the median. We hypothesized that this strategy would retain much of the efficiency and simplicity of randomized quicksort, while also exhibiting the lower variance of deterministic quicksort.

We started with sorting completely randomized arrays. As before, the smallest array size analyzed was 10,000 elements, and the largest, 1,000,000 elements; the size was incremented by 90,000 elements until it reached the maximum size. For each array size, the program averaged the runtimes of 100 trials of quicksort. To calculate the variance, the program ran quicksort on 100 completely randomized arrays of size 10,000. We repeated these experiments for partially and mostly sorted arrays.

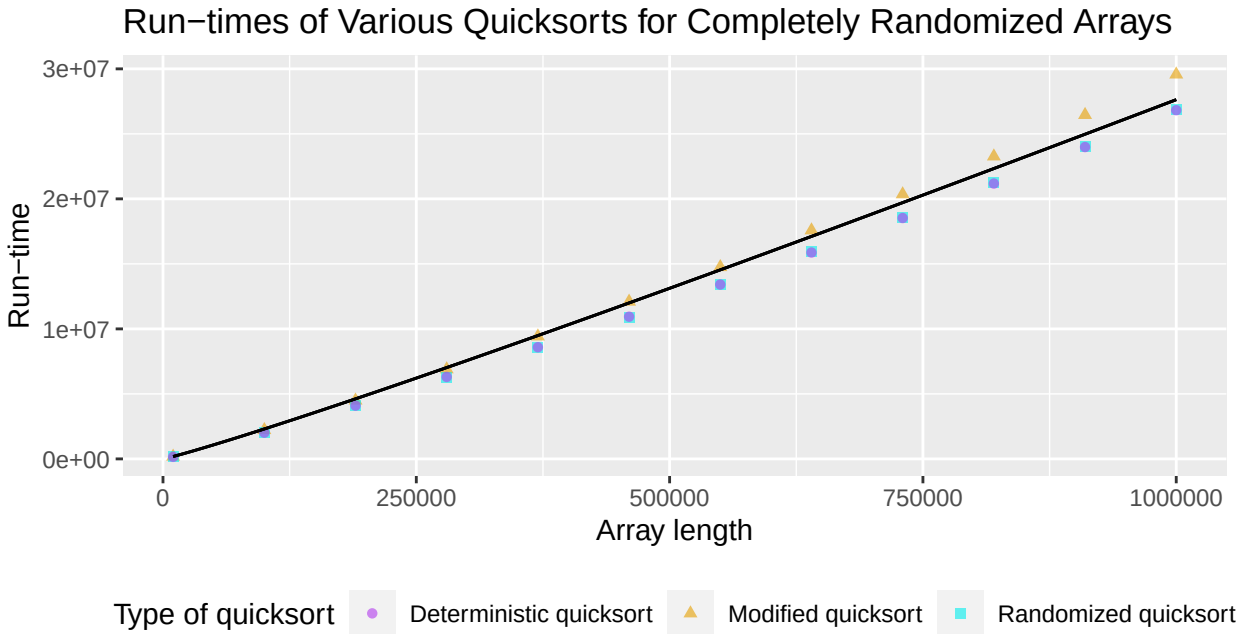
Results and Discussion

Analyzing Expected Run-time for Arrays of Varying Length

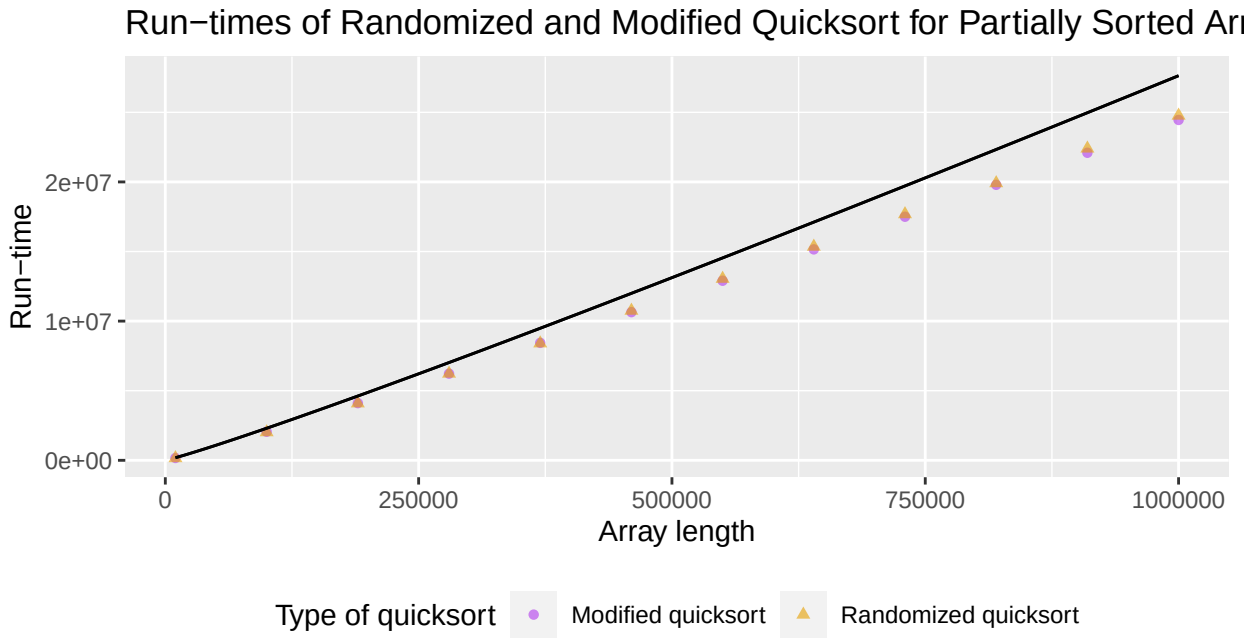
```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```



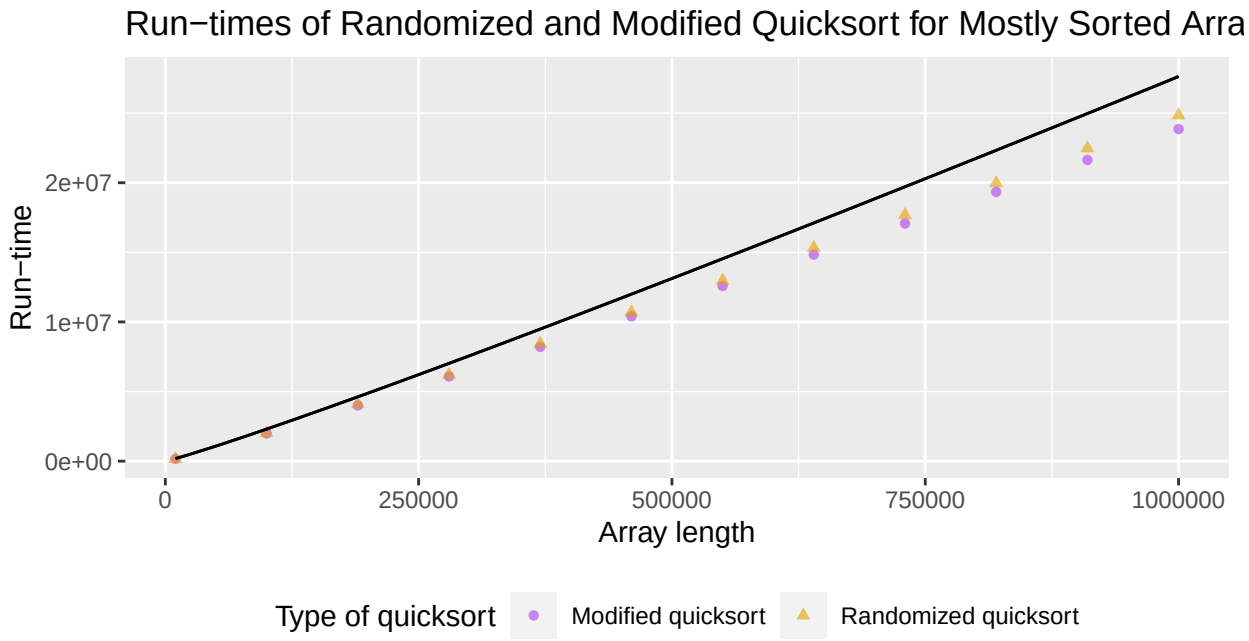

```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```




```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```



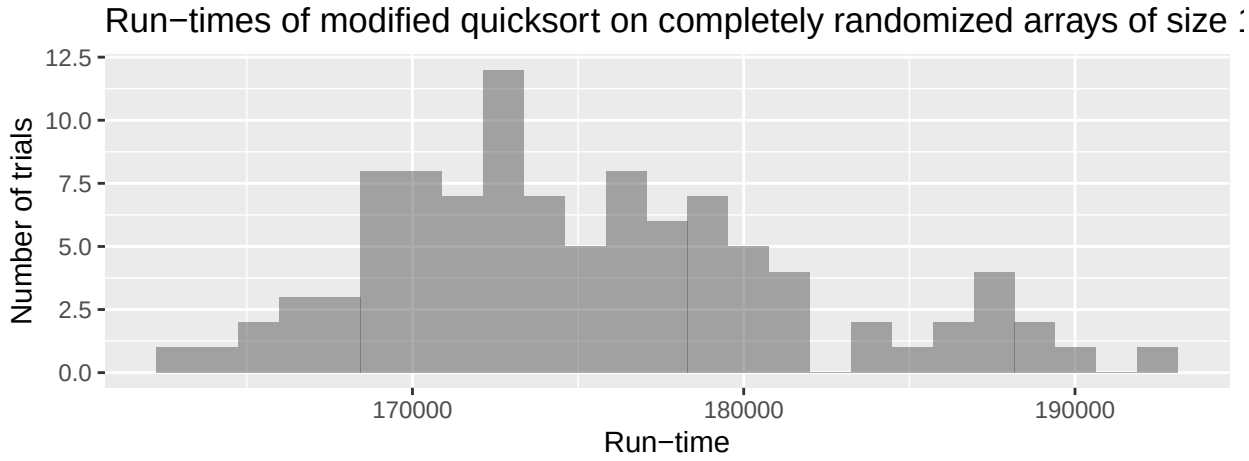

```
## Warning: Multiple drawing groups in `geom_function()`. Did you use the correct
## `group`, `colour`, or `fill` aesthetics?
```

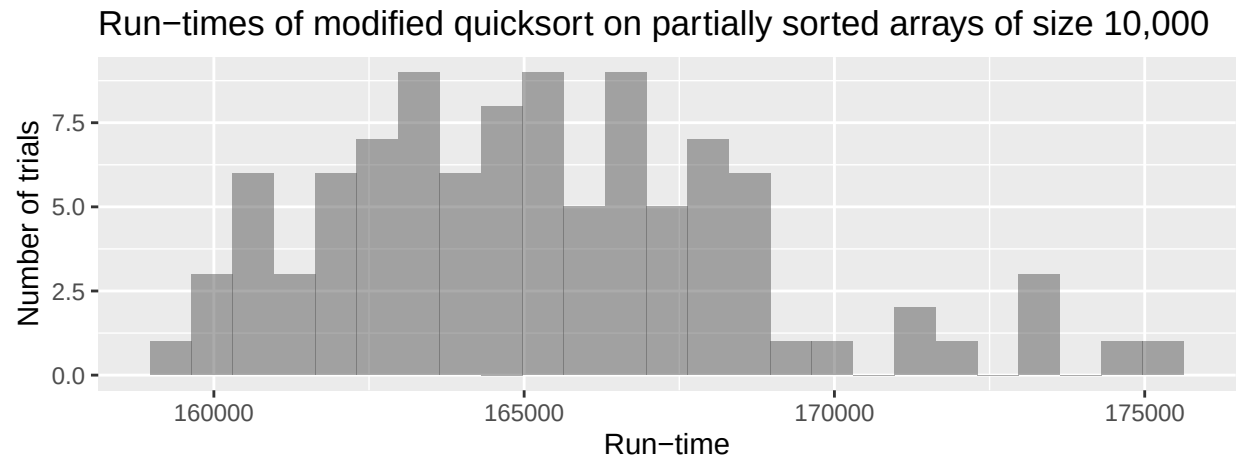


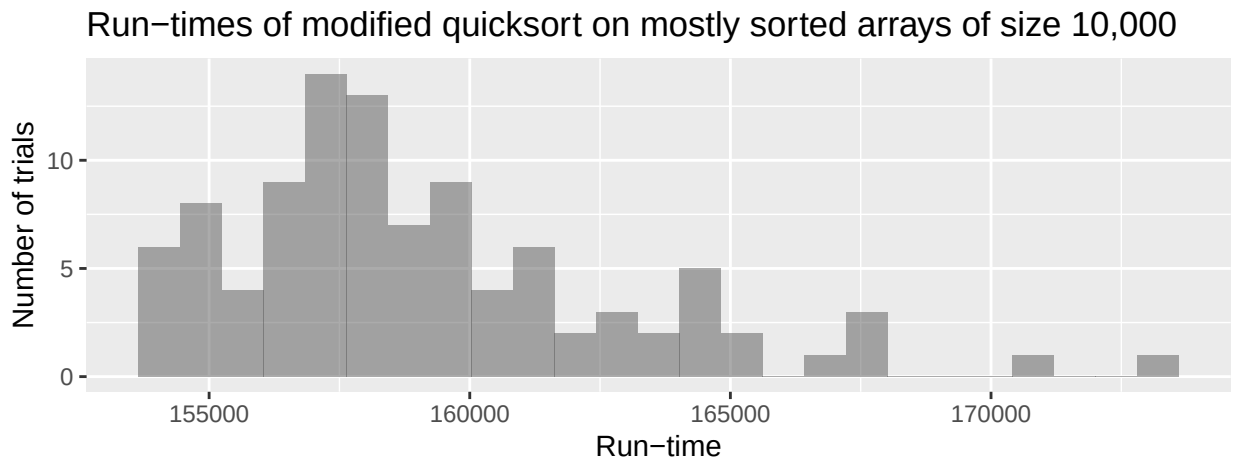
Over completely randomized arrays, the modified implementation was slower than the randomized and deterministic implementation in expected performance. This result is likely due to the increased number of comparisons required to choose the partition element; since we used bubble sort to sort the array of three random indices, each recursive call increased the runtime by nine comparisons. However, for partially and mostly sorted arrays, the modified implementation performed better in expectation than the other implementations. Notably, the average runtimes of the modified implementation for an array of a given size were lower than the average runtimes of the

randomized implementation, which in turn were lower than those of the deterministic implementation. The lower average runtimes for modified Quicksort were surprising, because we were unsure if the cost of additional comparisons would outweigh the benefits of potentially better partitions. Suggesting that, for certain inputs, the cost does not outweigh the benefits, our results can be tentatively interpreted to show the pitfalls of unbalanced subsets in Quicksort. However, we are unsure of precisely why the expected performance of modified Quicksort was more efficient than the other implementations for partially and mostly sorted arrays.

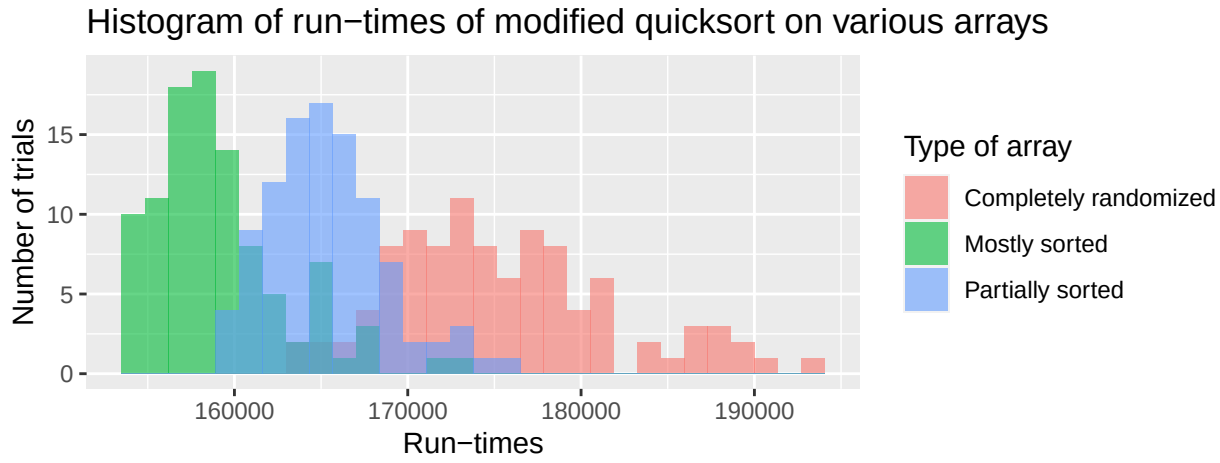
Analyzing the Spread of Runtimes for Arrays of a Given Size



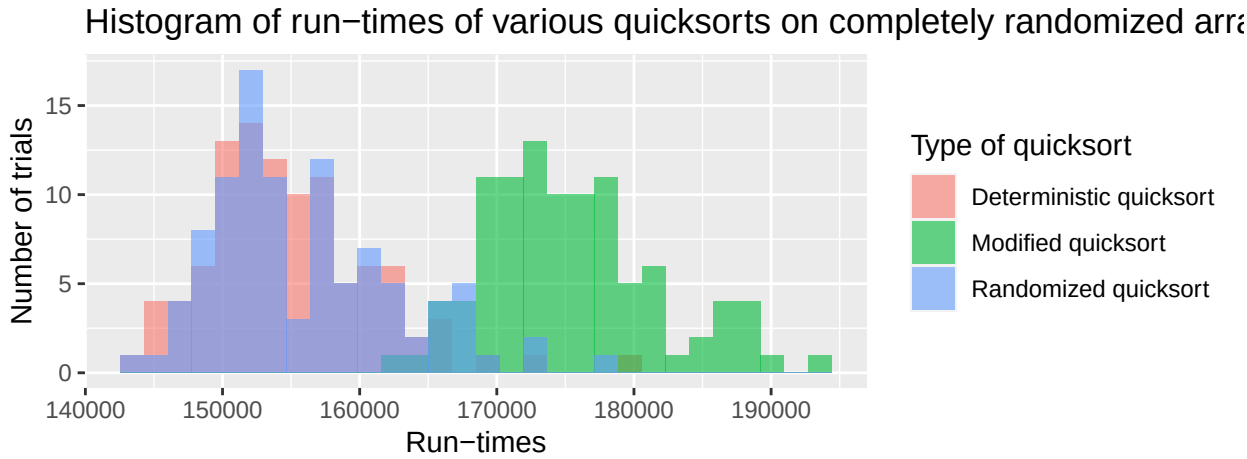




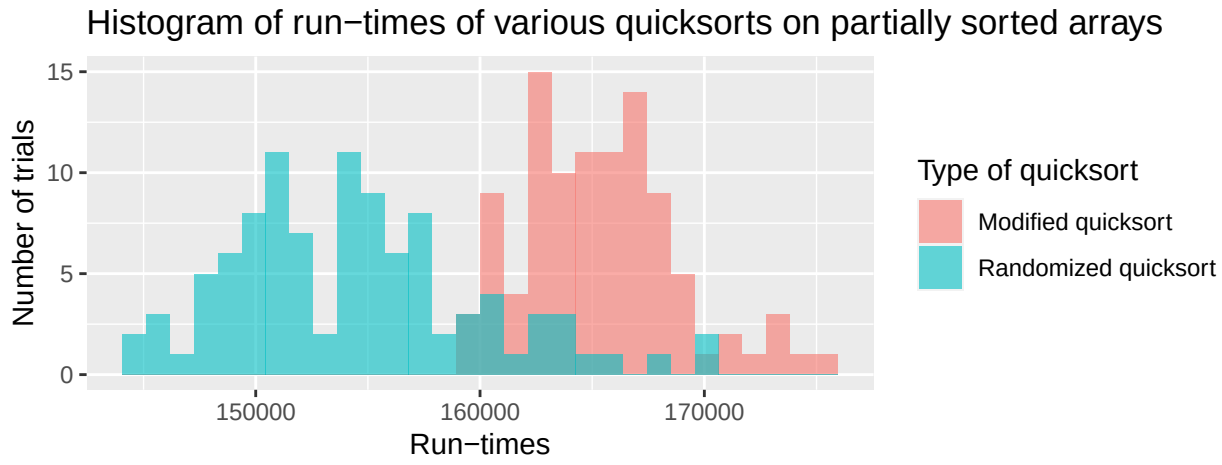

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



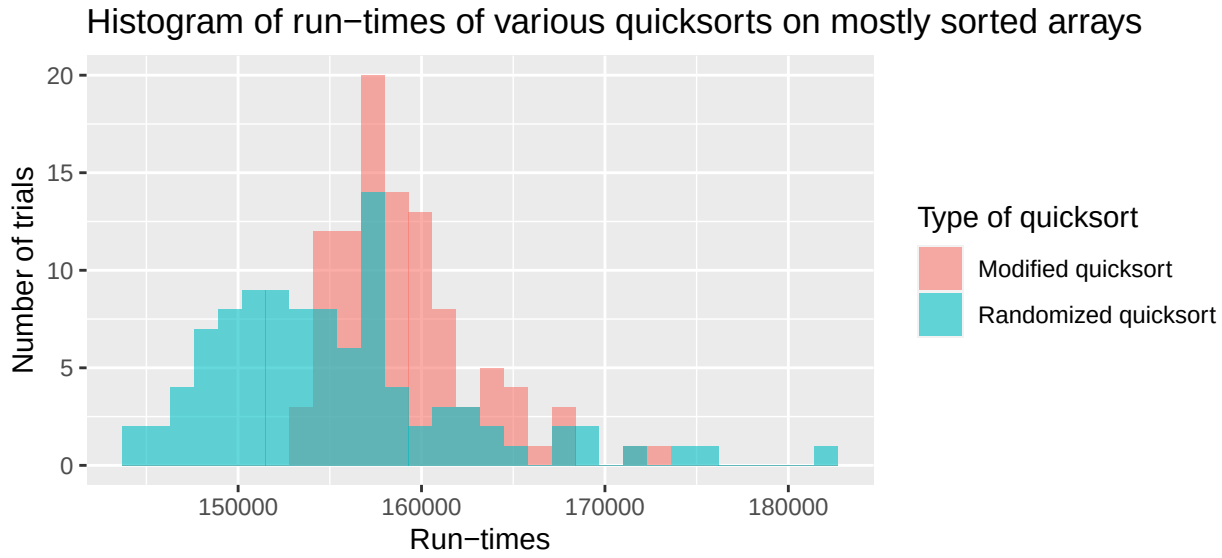

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Key Statistics of the Modified implementation

	Average	Variance	Squared coefficient of variance
Randomized	175404.41	4.020153123424242E7	0.001306656938366658
Partially sorted	165331.82	1.1608718633939397E7	4.246893712674323E-4
Mostly sorted	159113.94	1.6195375913681863E11	5.75900091780904E-4

For completely randomized and mostly sorted arrays, we also discovered that the modified implementation's normalized variance was lower than that of randomized quicksort, but higher than that of deterministic. Thus, our implementation was a slight improvement. The result can be attributed to the fact that randomized quicksort is more likely to choose "worse-case scenario" partition elements that result in unbalanced array subsets. The modified implementation accounts for this issue by providing two "backup" choices. Due to more consistent and efficient choices of the partition element, improved quicksort generally resulted in lower normalized variances, and lower average runtimes than partially and mostly sorted arrays.

Meanwhile, for partially sorted arrays, the modified implementation surprisingly had the lowest squared coefficient of variance. In our initial experiments, we encountered surprising results in this input as well, and we needed to explain this observation. However, in this case, the explanation above still aligns with this result. The modified implementation was simply more effective for this input, than it was for the others.

Conclusion

In conclusion, we discovered that randomization is a powerful tool, capable of decreasing the runtime of the Quicksort algorithm. Though it performed much like the deterministic implementation when run on completely randomized arrays, the randomized implementation was quicker in expectation on partially and mostly sorted arrays of a given size. Its generalizability is an asset, especially when its inputs' levels of sorting are not clear or predetermined. However, this implementation fails on one aspect: its variance. The variance of runtimes of the randomized implementation tended to be much higher than that of the deterministic algorithm. Should a user of Quicksort prioritize consistency in runtimes, over expected performance, the deterministic implementation may be a better choice.

After our initial experiments, we constructed a modified implementation of Quicksort that involved both randomization and determinism in its partition selection method. By combining characteristics of both implementations, the modified implementation was able to retain much of the efficiency of randomized Quicksort while reducing the variance of runtimes.

However, in our comparisons between randomized and deterministic Quicksort, we only analyzed one implementation of each. We therefore cannot infer that all randomized algorithms are more efficient than deterministic algorithms, because our observations and conclusions may be dependent on the particular implementations and algorithm that we chose to study. For example, a deterministic implementation of Quicksort that instead chose the median index as its partition element would likely perform much better than our deterministic implementation for partially or mostly sorted arrays, because it would tend to choose the median element in the array as the partition. Thus, we cannot generalize our results beyond the particular experiments that we ran.

This limitation means that our study of randomization in algorithms leaves much to be answered. To continue this study, we might repeat these experiments for randomized and deterministic implementations of a different algorithm or for different randomized and deterministic implementations of Quicksort. Additional research in this vein would allow us to broaden our conclusions and would help us assess whether or not the benefits—-simplicity, efficiency, generalizability—-and shortcomings—-high variance—-of randomization hold in general.

In addition, we had a number of lingering questions after completing our analysis. For a given array length, the randomized algorithm performed better for mostly and partially sorted arrays than for completely randomized arrays. On a human level, this result is logical, as it is quicker to sort arrays that are already partially sorted. However, from the level of code, the level of initial sorting should not impact runtime (the number of comparisons). We hypothesized that perhaps the recursive calls on sorted arrays collapsed more quickly down to the base cases, but we could not source this theory. Thus, we remain unsure about why the randomized algorithm's expected performance differed by level of initial sorting.

Furthermore, we noted that the spread of the runtimes of the randomized algorithm was greater than that of the deterministic algorithm (the normalized variance, the squared coefficient

of variance, was higher). However, for partially sorted arrays, the squared coefficient of the variance was lower; the spread of runtimes of the randomized algorithm was lower. This result may be due to the fact that partially sorted arrays may have various levels of initial sorting, which would lead to potentially very different sequences of partition choices under the deterministic algorithm. However, we did not have sufficient evidence to conclude this hypothesis with certainty.

Why did some experiments yield such surprising and confusing results? Even though these questions remain to be answered, our research demonstrates the fundamental benefits and shortcomings of a randomized algorithm versus a deterministic algorithm.