Analysis of Randomized Quicksort

William Duquette, Reece Quigley, and Khoi Viet Le

September 15, 2022

Abstract

In this paper, we will explore the probabilistic nature of the common sorting algorithms found in computer science. Some algorithms that will be discussed are BOGO-Sort, Quicksort, and Randomized Quicksort. BOGO-Sort serves as proof of concept of the probabilistic nature of sorting algorithms whereas Quicksort, and its various adaptations are more real-world ready algorithms that still carry some random aspects. We will discuss topics such as expected run times of each algorithm and the respective proofs. We will also discussed some of the worst-case scenarios and how they can arise in a random fashion. The paper will conclude with an analysis of the differences between Quicksort and Randomized Quicksort and how probability plays a role in the run times of each algorithm.

Sorting Algorithms

There are many ways to sort arrays in computer science. When most people think of these algorithms they tend to think of the classics: Insertion Sort, Selection Sort, and Quicksort. Although those are tried and true they tend to be slower than the most advanced sorting algorithms such as Merge sort and Heap Sort. But there are ways to improve the "classics" with randomization based methods.

Randomized algorithms are a relatively unique approach to sorting. Like the name suggests they rely on some level of randomness. Some randomized algorithms rely on randomization in a small way others are build entirely of that. An example of the latter is Bogo Sort.

Algorithm 1 BOGO-SORT, Parameters = Array

```
1: $n$ = length of Array
2: while IS-SORTED(Array) == $FALSE$ do
3:     SHUFFLE(Array)
4: end while
```

Bogo Sort is an extremely fascinating algorithm. Essentially, the code creates random shuffles of the array then checks the order. Bogo sort uses two main helper functions, the first being SHUFFLE, which creates random shuffles of the array:

Algorithm 2 SHUFFLE (Helper function for BOGO-SORT), Parameters = Array

```
1: $n$ = length of Array
2: for $i$ in length(Array) do
3:     $r$ = randomInteger$(0, n-1)$
4:     Array[$i$], Array[$r$] = Array[$r$], Array[$i$]
5: end for
6: return $TRUE$
```

The second helper function is IS-Sorted, which checks if the array is properly sorted:

Algorithm 3 IS-SORTED (Helper function for BOGO-SORT), Parameters = Array

```
1: $n$ = length of Array
2: for $i$ in length(Array) - 1 do
3:     if Array[$i$] > Array[$i+1$] then
4:         return $FALSE$
5:     end if
6: end for
7: return $TRUE$
```

In theory, the best case of this algorithm is one of the fastest ways to sort an algorithm. However, in practice it is significantly slower. There are obvious pitfalls with this approach. First, there is a very low probability that the array is sorted simply using a random shuffle. Assuming all values are unique there is only 1 correct sorting, meaning that the probability that the random order is correct is $\frac{1}{n!}$. This is an extremely low probability. Let us say that we are sorting an array with 10 unique elements. The probability that the randomized shuffle is in the correct order is $\frac{1}{3628800}$. Clearly, this is not an ideal sorting algorithm. In fact, the worst case runtime for this algorithm is $O(2^n)$, which is terrible when compared to the "classics." Even the average case runtime is slow. Remember the probability of the array being sorted after one random shuffling is $\frac{1}{n!}$. Which means that the expected number of permutations needed to sort the algorithm is $n!$. Also, remember it takes $\theta(n)$ to check if the shuffled array is correct (IS-SORTED). So to conclude it takes $\theta(n! \cdot n)$, which is not very fast when compared to the alternatives such as Insertion Sort, Selection Sort, and Quicksort. This is not to say that randomized algorithms are not useful. In fact, they are incredibly useful as we will show with Randomized Quicksort.

Quicksort

Before we can explain Randomized Quicksort we must first go over quicksort:

Algorithm 4 TRADITIONAL-QUICKSORT, Parameters = Array, low, high

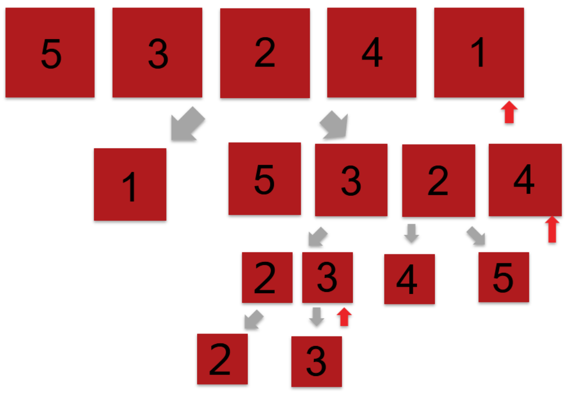
```
1: **if** low < high **then**
2:     $q$ = partition(A, low, high)
3:     quicksort(A, low, $q - 1$)
4:     quicksort(A, $q + 1$, high)
5: **end if**
```

Quicksort is a classic divide and conquer algorithm. Essentially, a pivot is picked and then all elements that are larger than the pivot are moved to the right of the pivot and those that are smaller than the pivot are moved to the left. Quicksort is then recursively called on the array of elements smaller than the pivot as well as on the array that contains the elements larger than the pivot. Since we called Quicksort recursively, it calls partition and sorts in two sub arrays the same way as above. The crucial function to making quicksort work is partition. Below is the psuedo code for partition:

Algorithm 5 TRADITIONAL-PARTITION, Parameters = Array, low, high value

```
1: $x$ =A[high]
2: $i$ = low − 1
3: for j in range low to high do
4:     if $A[j] \leq x$ then
5:         $i = i + 1$
6:         $y = A[i]$
7:         $A[i] = A[j]$
8:         $A[j] = y$
9:     end if
10: end for
11: $z = A[i + 1]$
12: $A[i + 1] = A[high]$
13: $A[high] = z$
14: return $i + 1$
```

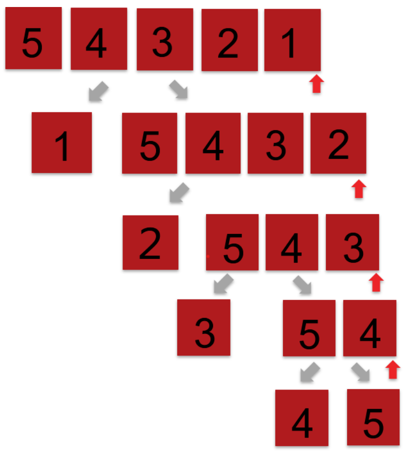
In order to better understand this let us walk through a quick example of Quicksort, on the array: $[5, 3, 2, 4, 1]$. Like was mentioned above quicksort is a divide and conquer algorithm, meaning that this example can be broken down visually into a tree structure:



Now it is time to analyze the runtime of quicksort. The expected runtime of Quicksort is $\theta(nlogn)$. However, its worst case is still $O(n^2)$. This is when it is important to talk about what would cause Quicksort to have its worst case runtime. There are orders of arrays that guarantee that Quicksort has its worst case runtime. For example, the array:

$$[5, 4, 3, 2, 1]$$

would cause the worst case. We can see our tree below (this tree can be thought of as the structure of our recursive calls:



This array would be Quicksorts worst case because when we pick the last element in the array, in this case 1, we would have to move everything to the other side of our pivot. This is bad for a view reasons, the array that holds elements that are smaller than 1 would be empty, which is unbalance our tree. Second, when we call Quicksort on the elements larger than 1, and pick a pivot, it will be 2, which will cause everything to move the right of 2, and so on and so forth. Abstractly, we would have to move all n remaining elements in the array to the other side of the pivot, which takes $O(n)$. Then we would have to do that process n times, hence the worst case being $O(n^2)$.

Worst Case Runtime of Randomized Quicksort and Quicksort

As mentioned above the worst case run time of Quicksort is $O(n^2)$. Below is the proof for this run time. Additionally, this proof also holds for Randomized Quicksort's worst case runtime. In order to prove the Big O runtime of this algorithm, we must first define what an asymptotic upper bound is.

Definition 0.1. $O(g(n)) = \{f(n)$: there exist a positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$

We will use the substitution method for this proof.

The recurrence equation is $T(n) = T(n) + T(n-1) + n$.

Base Case: $T(1) : 1 + 0 + 1 = 2$

We need a c such that: $2 \leq c(1)^2$. In this case, $c = 2$.

Our best guess is $T(n) = O(n^2)$

The induction hypothesis is $\forall k < n, T(k) \leq O(k^2)$.

$$T(n) = T(n-1) + T(n) + n \leq (c(n-1))^2 + (cn)^2 + n$$

$$= (cn - c)^2 + cn^2 + n$$

$$= cn^2 - c^2 + cn^2 + n$$

$$\leq (c - c^2 - c)n^2 + n$$

$$\leq cn^2 \text{ when } c \geq 5$$

When $n_0 = 3$ and $c \geq 5$ then $0 \leq T(n) \leq c * g(n), \forall n \geq n_0$. Therefore $T(n) = O(g(n))$.

Best Case Runtime of Quicksort

In the best case, the quicksort algorithm would choose the pivot to be the median every time. Thus, the recurrence of the runtime of quicksort would be

$$T(n) = T(\frac{n}{2}) + cn$$

This is due to the fact that the pivot is the median, so when we use the partition function, we would divide the array into two halves $(2T(\frac{n}{2}))$ in linear time (cn). Solving the recurrence, we will get $T(n) = O(nlog(n))$.

Average Case Runtime of Quicksort:

So how about the average case runtime of quicksort? How can we solve it if we do not know what is the pivot that we will be choosing? Well, in the average case, lets assume that we are dealing with an array with size n. Since we are in the average case, it is fair to assume that all of these n elements are randomly assigned to their places. Given such assumption, any elements in the array can be the last elements with the probability of $\frac{1}{n}$, and thus, they are equally chosen to be the pivot.
So the recurrence $T(n)$ can be divided into $T(n-1) + T(0)$, $T(n-2) + T(1)$, ..., $T(0) + T(n-1)$ and every scenario happens with the probability of $\frac{1}{n}$. Thus, the recurrence would be formally expressed as

$$T(n) = \frac{1}{n}(\sum_{i=0}^{n-1}(T(i) + T(n-1-i))) + cn$$

As mentioned before, the term cn represents the time to perform the partition function. Solving the above recurrence is very long, but eventually, $T(n) = O(nlog(n))$, which is the average case of quicksort.

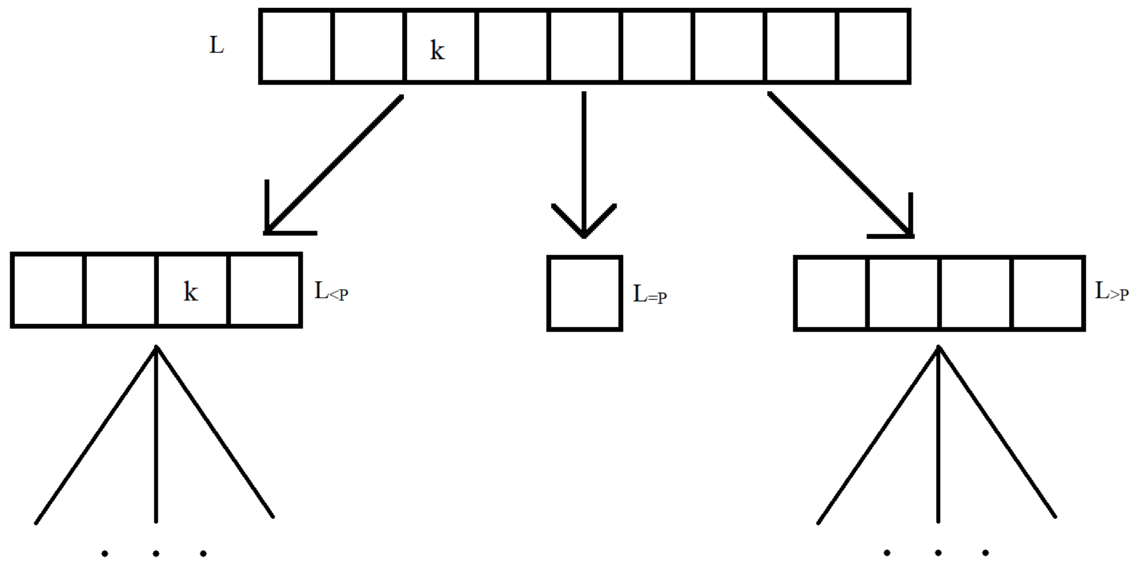
Randomized Quicksort

The only difference of Randomized Quicksort is that Randomized Quicksort chooses the pivot randomly instead of choosing the pivot as the last element of an array. Before we proceed further to the analysis, we will firstly define the pseudocode for the Randomized Quick Sort:

Algorithm 6 Randomized Quick Sort $(L, low, high)$:

```
1: if $|L| \leq 1$ : then
2:     return $L$
3: end if
4: pivot $\leftarrow$ rand$(1, |L|)$
5: $P \leftarrow L$[pivot]
6: $L_{<P}$, $L_P$, $L_{>P} \leftarrow$ partition $(L, P)$
7: $L_{<P} \leftarrow$ Randomized Quick Sort $(L_{<P})$
8: $L_{>P} \leftarrow$ Randomized Quick Sort $(L_{>P})$
9: return $L_{<P} + L_P + L_{>P}$
```

The line 6 of the pseudocode partitions into three parts as $L_{<P}$ represents the list containing every element in L that is less than P, L_P represents the list containing only P, and $L_{>P}$ represents the list containing every element in L that is bigger than P. Line 7 and 8 performs the algorithm recursively on the two lists $L_{<P}$ and $L_{>P}$, and at the end, the algorithm returns the sorted list.



The above picture illustrates a tree containing the first few steps of the algorithm, where a list L is divided into three sub-lists, and the process continues until the length of every sub-list is one, i.e. the list L is sorted. In the tree above, for an element k in list L, we define P_k to be the path that the element k takes from the root, i.e. list L, to its leaf where a sub-list only contains k. In addition, we also define $|P_k|$ to be the number of edges in P_k. To calculate the run time of our algorithm, we would sum up all the length of all paths P_k of every element k in the list.

We also define different types of edges in the tree. Specifically, let $|A|$ be the length of a list A, then for an element k,

- An edge is black if $k \in L_P$
- An edge is called blue if $k \in L'$ and $|L'| \leq \frac{1}{2}|L|$
- An edge is called red if $k \in L'$ and $|L'| > \frac{1}{2}|L|$

For the above defined colored edge, we will define $|P_k(\text{black})|$, $|P_k(\text{blue})|$ and $|P_k(\text{red})|$ as the total number of black edges, blue edges and red edges in P_k, respectively.

Analysis of the runtime of the Randomized Quick Sort:

For every element k in a list, $|P_k| = O(\log n)$ in expectation.

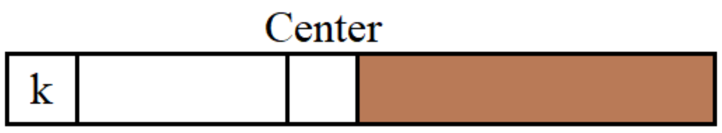
Proof. Let n be the length of a list. Based on the definition of colors of edges we defined above, we have that for any path P_k, the total number of edges in P_k is equal to the sum of all black edges, blue edges and red edges in that path. In other words,

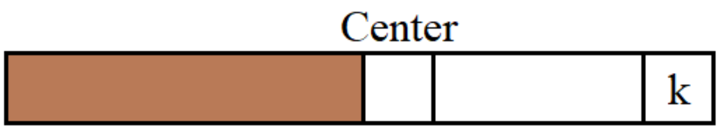
$$|P_k| = |P_k(\text{black})| + |P_k(\text{blue})| + |P_k(\text{red})|$$

Now, we only have to consider every type of edges in that path:

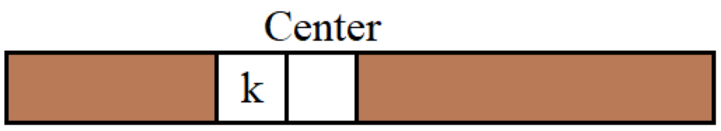
- Consider the number of black edges in P_k, it is at most one because only the last edge connected to the list containing only k can be black or k is divided into its own list, when there cannot be a black edge.
- Since the length of the list connected to a blue edge is at most a half of the original list, let the number of blue edges be x. The number of blue edges can be calculated when we reduce the size of the list by half until it reaches 1, or $\frac{n}{2^x} = 1$, meaning that $x = log(n)$. Thus, we will have at most $log(n)$ number of blue edges.
- For the number of red edges, based on the definition, it is hard to define the exact number because the length of the sub-array is more than half of the length of the original list. Therefore, it would be hard to estimate what is the expected number of red edges.

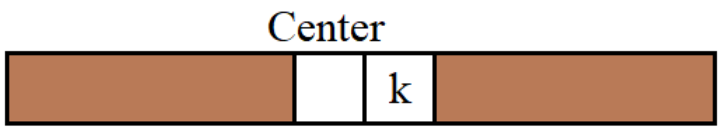
Considering the red edges, we will be using probability to find the number of red edges in a path P_k. To calculate the probability of having a red edge, we will consider the following scenario on a sorted list:





If we let k to be the smallest element or largest element in the list, then we can choose the pivot to be any element in the shaded region to get a red edge on k. Therefore, the probability is roughly a half. However, if we move k nearer to the center:

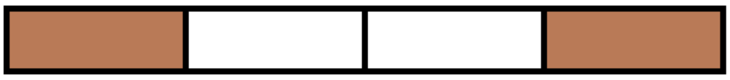




Then, for this case, the probability to have a red edge on k is roughly one because we can choose the pivot to be any element in the shaded region and k will be in the bigger partition. Therefore, the probability of having a red edge is not constant, and we have to find a different way to make it consistent. Therefore, to make the expected number of red edges to be consistent, we will redefine the blue edges and red edges as:

- An edge is called blue if $k \in L'$ and $|L'| \leq \frac{3}{4}|L|$
- An edge is called red if $k \in L'$ and $|L'| > \frac{3}{4}|L|$

Based on this definition, using the same reasoning, we can show that the number of blue edges in P_k is at most $\log_{\frac{4}{3}} n$, and the number or black edge is still at most one. Considering red edge:



We will divide the sorted list into 3 quantiles, and for every element k in that list:

- If we choose the pivot to be in the shaded region, there will be one blue edge and one red edge.
- If we choose the pivot not to be in the shaded region, there will be guaranteed two blue edge no matter where k is in the sorted array because the partition will always divide into two sub-lists of size less than $\frac{3}{4}|L|$.

Therefore, regardless of the element in the list, the probability that a red edge appear will be at most $\frac{1}{2}$. Note that not element has the same probability $\frac{1}{2}$ of having red edge during a particular partition. If element k is not in the shaded area, then the probability of having red edges is $\frac{1}{2}$ because if the pivot in the shaded area, k will have a red edge. However, if k is not there, the probability will be less than $\frac{1}{2}$. For the sake of bounding the runtime of the algorithm, we would assume it to be $\frac{1}{2}$ for every element.

Now, we can bound the number of red edges based on the number of blue edges. Let X_i be a random variable equal to the number of red edges between the i^{th} blue edge and the $i+1^{th}$ blue edge. The probability of having a red edge for every element k is $\frac{1}{2}$, so what is the expected number of red edges until we see a blue edges? Well, this is basically a geometric random variable. Therefore, we will have

$$\mathbb{E}[X_i] \leq E[geo(\frac{1}{2})] = \frac{1}{\frac{1}{2}} = 2.$$

Therefore, in a path P_k, since the number of blue edges is at most $log_{\frac{4}{3}}n$, the number of red edges will be at most $2\log_{\frac{4}{3}} n$, and thus,

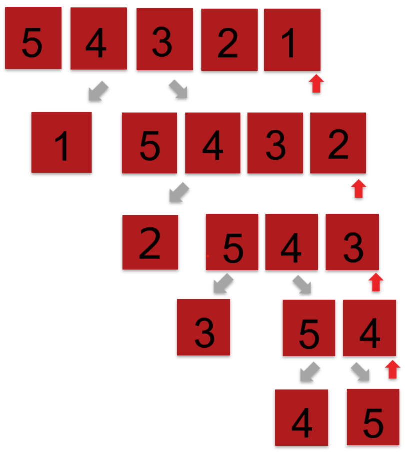
$$|P_k| \leq 1 + \log_{\frac{4}{3}} n + 2\log_{\frac{4}{3}} n = O(\log n)$$

$\square$

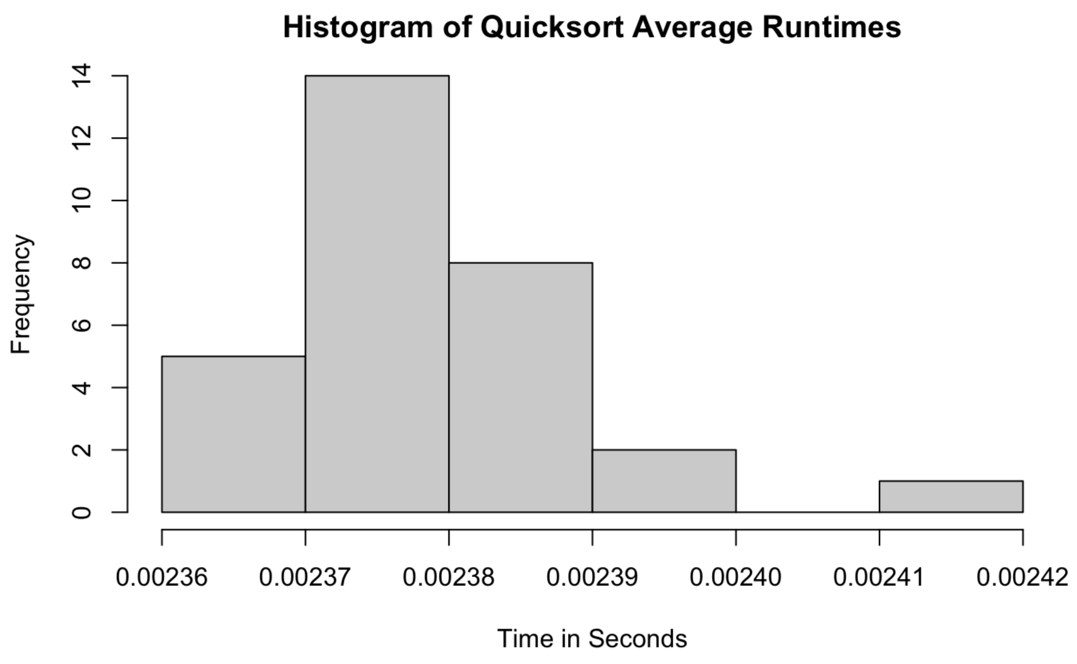
Based on the tree we shown above, the expected runtime of the Randomized Quick Sort can be also expressed as $O(\sum_{k\in L} |P_k|)$, and we know that $|P_k| = O(\log n)$ in expectation for every element k. Therefore, the runtime of the algorithm is $O(n \log n)$.

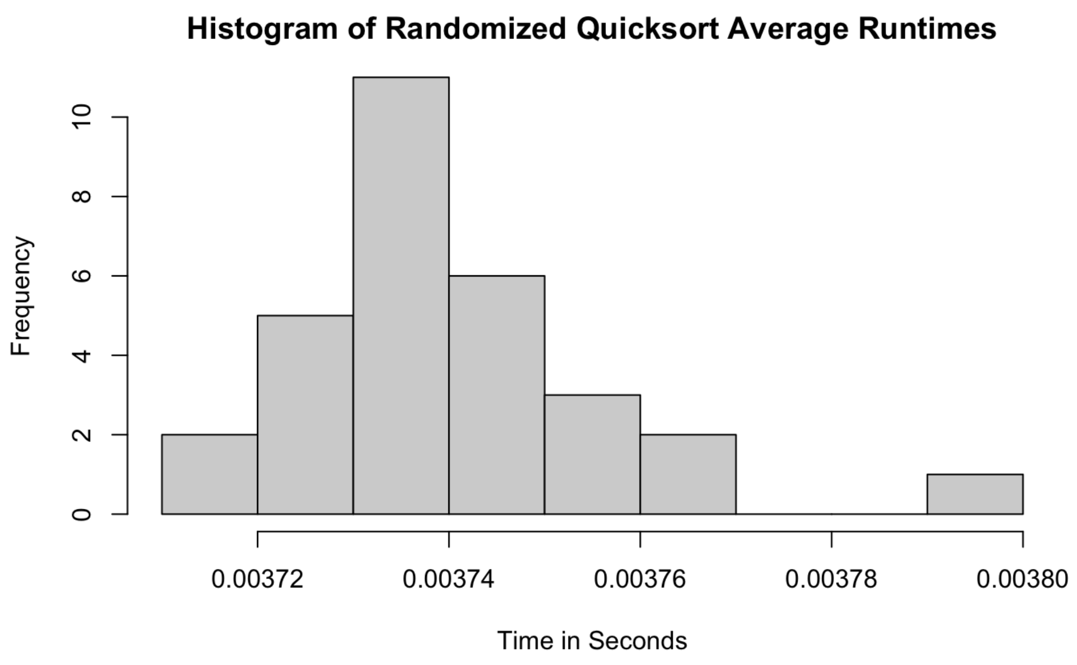
Comparing Randomized Quicksort and Traditional Quicksort

One of the aspects of these algorithms that we were interested in exploring is the differences between them depending on the structure of a given array. In a real world scenario, one might have some insights on the typical structure of their data and wonder if one algorithm performs better than the other. A good example of this scenario would be a sorted array that is backwards, identical to the one discussed above.



First, we decided to do this comparison purely on a run-time basis for a randomized array to see if there would be any differences. The strategy would be to develop a way to collect sample means and apply the central limit theorem to those means to understand the differences in each distribution for the two algorithms. The histograms of the average run times are below.

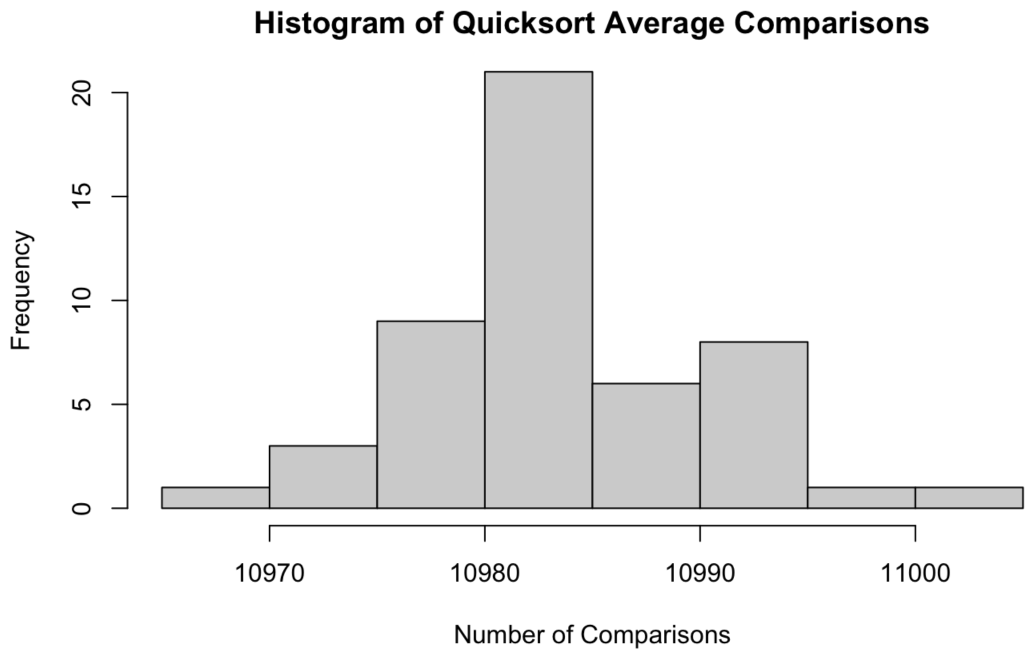


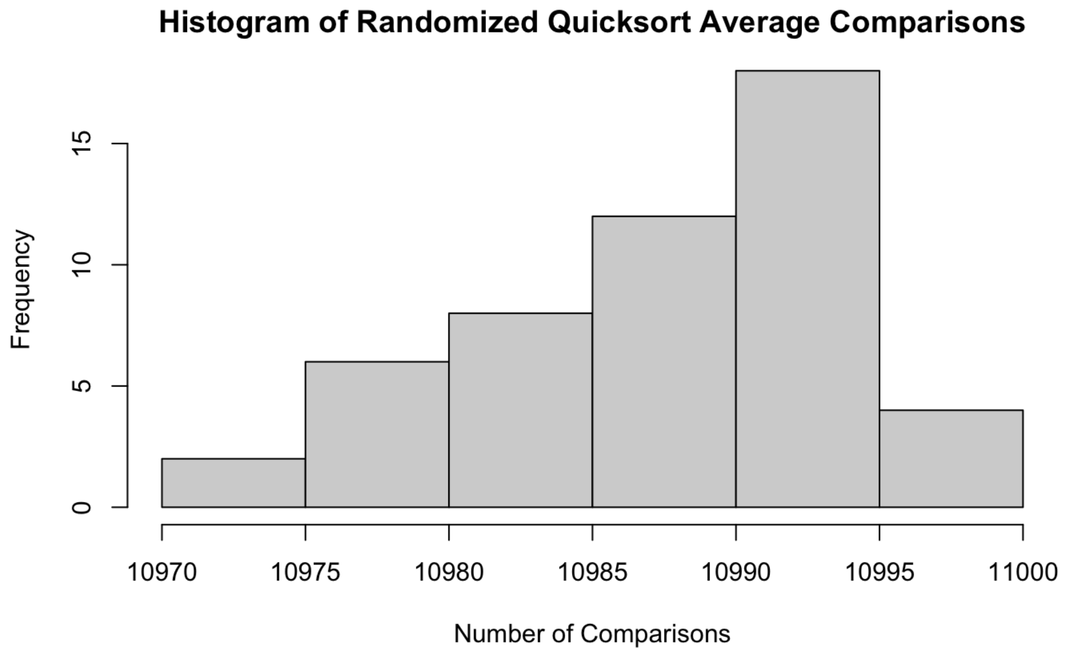


There was a difference in means of about 0.00136 seconds between each group with the quicksort algorithm being faster by that amount. Initially, this was an extremely surprising result since common literature would have predicted the opposite. But, upon further inspection, it makes sense. This is due to the fact that the randomized algorithm requires the use of a random number generator to pick a random pivot for each sort. This adds compute time ($logn$), resulting in a slower sort time. The random number generator randomly picks a number from a specified uniform distribution, and this added compute time makes the algorithm slower in the real world setting.

Another interesting issue was the lack of normality in each histogram. Considering our procedure and our usage of the central limit theorem; we should have be left with a normal distribution, but we were not. We even tested this with 50 sample means and still did not receive a normal distribution. Upon closer inspection, we obviously violated the conditions of the central limit theorem in some sort of way; mostly likely not having independent or identical distributions between the runs.

The follow up procedure was to determine a way to measure performance of the algorithm outside of compute time. Then, we could hopefully actually use the central limit theorem and have a way of more theoretically evaluating the probabilistic nature of each algorithm. We decided to compare the number of comparisons each algorithm used using an identical method as discussed previously. The graphs of the procedure are below.





As one can see from the histograms however, we still did not reach complete normality for the randomized quicksort. Again, some sort of assumptions of the central limit theorem must have been broken here, but that is outside the scope of this analysis. The randomized algorithm on average required about 3.55 more comparisons than the standard quicksort. This is surprising, but the difference is almost negligible. It is important to mention that this is all for essentially completely random data arrays. The moment that structure is added into the data array, randomized quicksort will tend to outperform as discussed earlier. But randomized quicksort is actually typically slower for completely random data.

It's likely possible to prove this on a larger scale, but for a completely random data array, it is essentially identical if you pick the pivot to be the last index or simply a random index. In the case where the last pivot is chosen, you are guaranteed to solve a single inversion in the array. If a random pivot is chosen, you can solve more than one, but it is also possible to solve none. This is likely some sort of symmetric distribution and thus cancels out to solving one on average. Either way, for a random array, it doesn't make a difference for which pivot you choose. Therefore, the compute time required to choose a random pivot is truly just wasted.

Conclusion

To conclude, we have discussed some of the most common sorting algorithms and their respected average case run times. Each average case had its respective proof which were constructed. We also did a more practical analysis on the real-world run times of each algorithm and how they compare against one another. Despite the common literature available, regular quick sort appears to on average be the more efficient algorithm for random data.

References

https://arxiv.org/pdf/1006.4063.pdf

https://dl.acm.org/doi/pdf/10.1145/234313.234327

https://www.sciencedirect.com/science/article/pii/0166218X9190086C

GitHub

All code that is used for this paper can be found at: Link