Complexity

Introduction

In this practical we will be looking at how the performance of a vanilla quicksort implementation varies according to the ‘sortedness’ of a list. This report should lay out how I implemented quicksort, a brief discussion of measures of sortedness and the input lists I will be testing, different measures for its efficiency and an analysis of test data of quicksort for lists of different sortedness.

Design

Quicksort

The implementation of quicksort we’ll be examining here is the not-in-place version that makes use of new, smaller and smaller lists before recombining rather than the in-place version that makes of use of pointers to the larger list. The main factor that will influence the speed of quicksort is its choice of pivot at each splitting of the list as every element will be compared to that pivot. I will examine last element pivot.

Sortedness of a List

There are a number of methods to define how ‘sorted’ a list is:

Example List: $W_1 = \{ 5, 8, 10, 9, 11, 6, 7,4\}$

1. **Distance.** The distance of an swap of two elements of a list. A larger swap is more disordered than a shorter one (Estivill-Castro, Wood 1992). e.g when looking at (5, 6) it would have a Dis(W_1) of 5.
2. **Max.** The maximum distance an element has to travel to reach its sorted position (Estivill-Castro, Wood 1992). e.g 4 has to travel seven positions to reach its sorted position.
3. **Exchange.** The minimum number of exchanges required to sort a sequence (Estivill-Castro, Wood 1992). Also known as the ‘edit distance’ when applied to strings.
4. **Rem.** The the minimum number of elements that must be re-moved to obtain a sorted subsequence (Estivill-Castro, Wood 1992).
5. **Runs.** The number of boundaries between ascendings runs of numbers within the list (Estivill-Castro, Wood 1992). e.g. $W_1 = \{ 5, 8, 10 \,||\, 9, 11|| \,6, 7|| \,4\}$ therefore it has 3 runs.

This is important when considering test cases for the quicksort algorithm in order to acquire a more precise scientific definition of ‘sorted’ for testing purposes. For the purposes of this report I decided to use the Removal and Exchange Metrics.

I used exchange because it is simple to create lists of varying number of exchanges, however it is difficult to determine the *minimum* number of exchanges required and thus the sortedness of the list.

The benefit of the removal metric is that there is an algorithm that can find the longest ascending subsequence in O(n log n), this allows us to calculate the sortedness of a particular list (by removal value) by finding the longest ascending subsequence and dividing its length by the total length of the list. This is because the longest ascending subsequence will be the created when removing the minimum number of elements possible.

Measuring Quicksort

I chose to use number of comparisons made to measure quicksort. This is because this should be a system agnostic way of finding out how long quicksort takes to work, and it provided useful information even in non-pathologic cases of quicksort.

I did consider using system time however that had the disadvantage that system time would be affected by a variety of factors such as how powerful the processor was and the what other processes running at the same time. In addition, the size of the list had to be very large in order to get useful (non-zero) measurements for non-pathological cases of quicksort that make them harder to compare. The main advantage being that system time would give a good indicator for how quicksort actually worked *in-practise* rather than an abstract measure.

Implementation

Input Lists

Exchange Metric

To test the exchange metric, I first create a sorted list of 4000 elements. Then I randomly swap x elements in the arraylist (starting at 0, incrementing by 10, then by 100 for values higher than 100 at each iteration up until the x = list length). Then I measure how many comparisons the quicksort algorithm makes when sorting the list. I repeat this process 20 times for different lists at each value of x and record the number of comparisons done each time.

I repeat this process for different lists because when swapping elements, the same element could be swapped multiple times so the resulting list isn’t exactly the number of swaps away from a sorted list. By taking multiple results I can get an average that should be closer to the true value of a

Removal Metric

To create lists of different sortedness according to the removal metric, there are two different methods I used:

1. Remove random elements from the sorted list and append them to the end.
2. Remove random elements from the list and put them at the front of the sorted list.

Analysis

Exchange

The exchange metric tests show that quicksort has a two primary pathological cases, a sorted list and an anti-sorted list. This is because in normal cases (Figure 1.) the sublists get progressively smaller and each pivot is only compared to elements in its own sublist, reducing work.

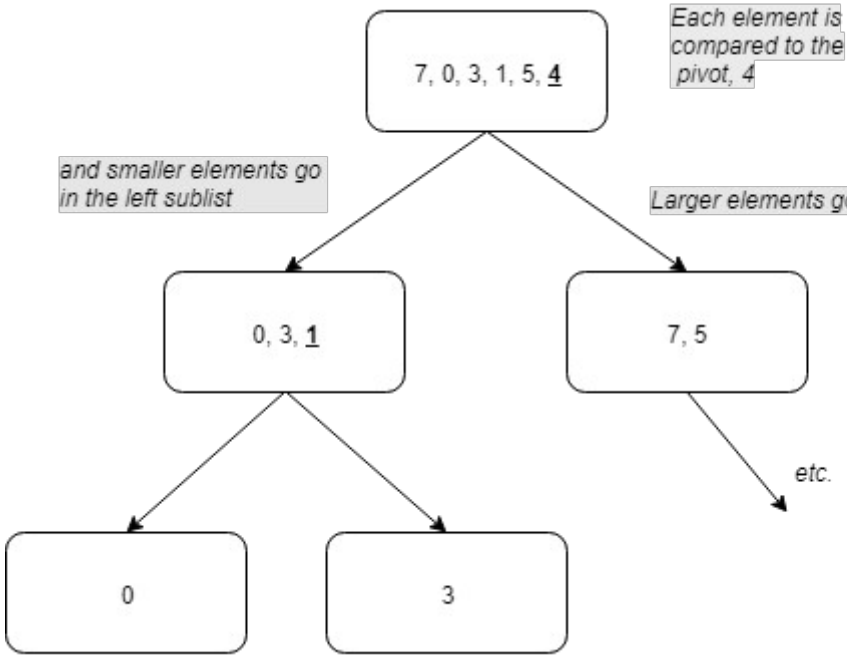


Figure 1. *Normal Case Quicksor*t

However in the case where the list is sorted and the pivot is the last (or first) element, every element will be added to the smaller sublist on each occasion. This means that the list being compared at each stage is not getting any smaller and n(n-1)/2 comparisons will have to be made (assuming no duplicate elements), giving a $O(n^2)$ or quadratic time complexity (Illustrated Figure 2.). Anti sorted lists perform similarly except the values are added to right (‘more than’) than sublist each time.

Using the exchange metric, I determined that there only needs to be a very small amount of unsortedness for the algorithm’s efficiency to increase. Just by swapping 10 elements (Appendix B), the number of comparisons decreased from 6.5 million to 2 million, and when looking at Appendix A there is a clear exponential decrease in the number of comparisons the more inversions that are.

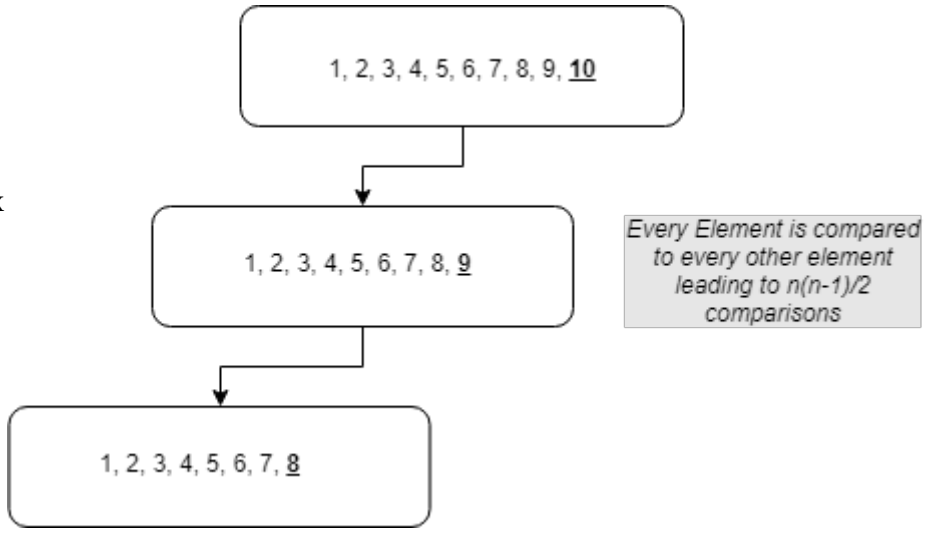


Figure 2. *Sorted List Case (54 Comparisons)*

I think this is because each swap makes it more and more likely that a value smaller than the last (and largest) element will be chosen as pivot. As soon as a non largest value is chosen as the pivot, the list is split into two parts drastically reducing the number of comparisons being made (Figure 3.)

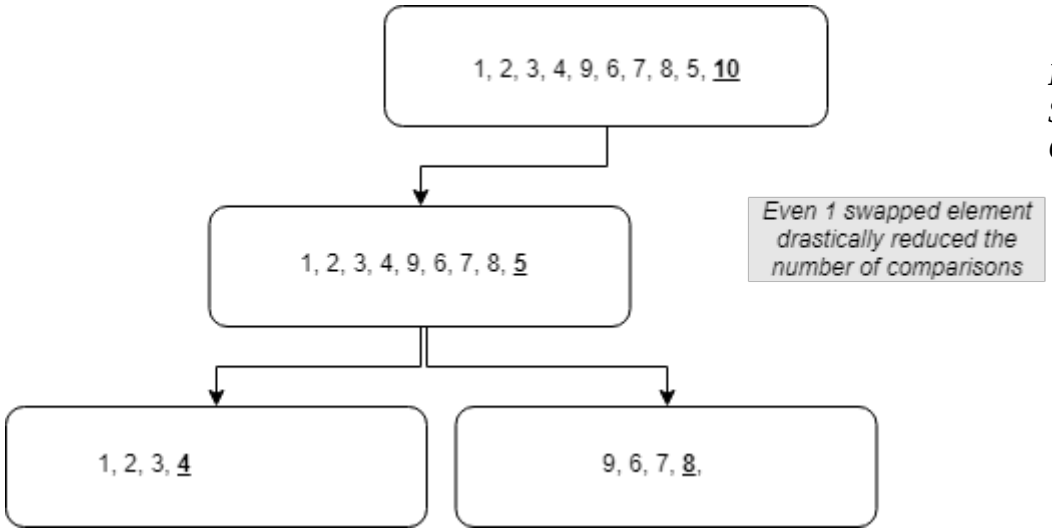


Figure 3. *One Swap Case. (34 Comparisons)*

This also explains to an extent why Appendix A isn't a smooth asymptote. The increase in efficiency will depend on how close the pivot is in value to the median, as the median would split the list into two equal smaller halves. To illustrate, if in Figure 2. 8 and 9 had been swapped instead there would be 46 comparisons, because one of the sublists would be much longer than the other, not cutting down on as many comparisons.

Removal

Random Elements at Front

When putting random elements in the list to the head of the list, quicksort performance increases linearly with unsortedness (Appendix C). This is due to the last element still being the pivot. As performance will only improve once the pivot isn't the largest value in the list, this will only occur once it reaches the part of the list where random elements have been added.

Random Elements at the Back

When putting random elements to the end of the list, quicksort performance increases exponentially (Appendix D) as these random elements have a high chance of being smaller than the largest value and thus the list is split into smaller sublists very quickly.

Average Sortedness

We can calculate the average sortedness of the lists used for the above (removal only) testing by finding the longest ascending subsequence algorithm and dividing its length by the total length of the list. This shows that regardless of method used to unsort the list they both are almost equal in unsortedness, yet the performance of quicksort improves at different rates (Appendices E & F).

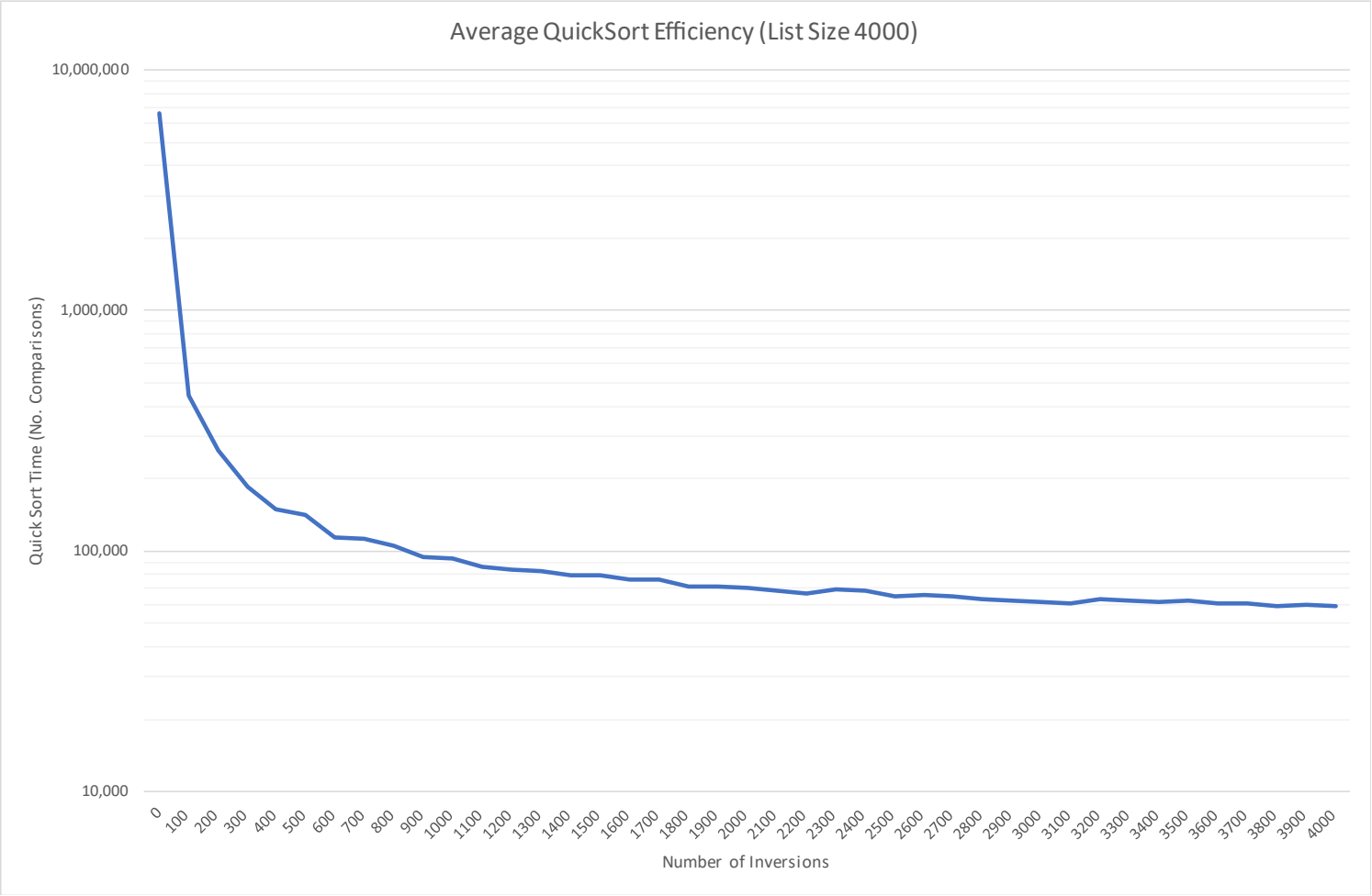
Conclusion

For unoptimised quicksort, performance, increases exponentially with unsortedness in most cases. The pathological cases being a fully sorted list and an anti-sorted list. Also by examining the Removal metric we found that even lists with similar sortedness might get different performance improvements from unoptimised quicksort.

Appendix

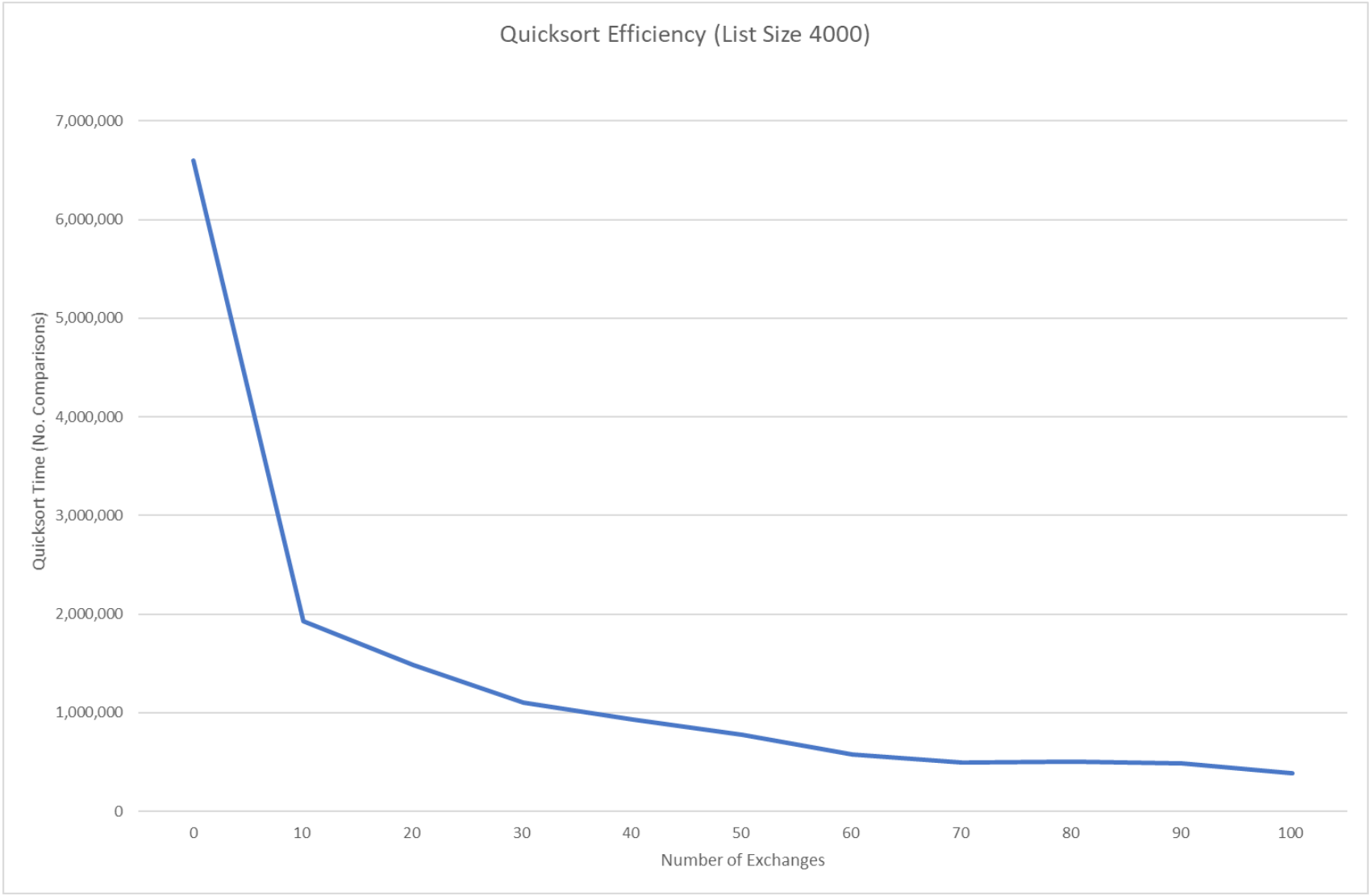
Appendix A

Number of Comparisons against Differently Sorted Lists



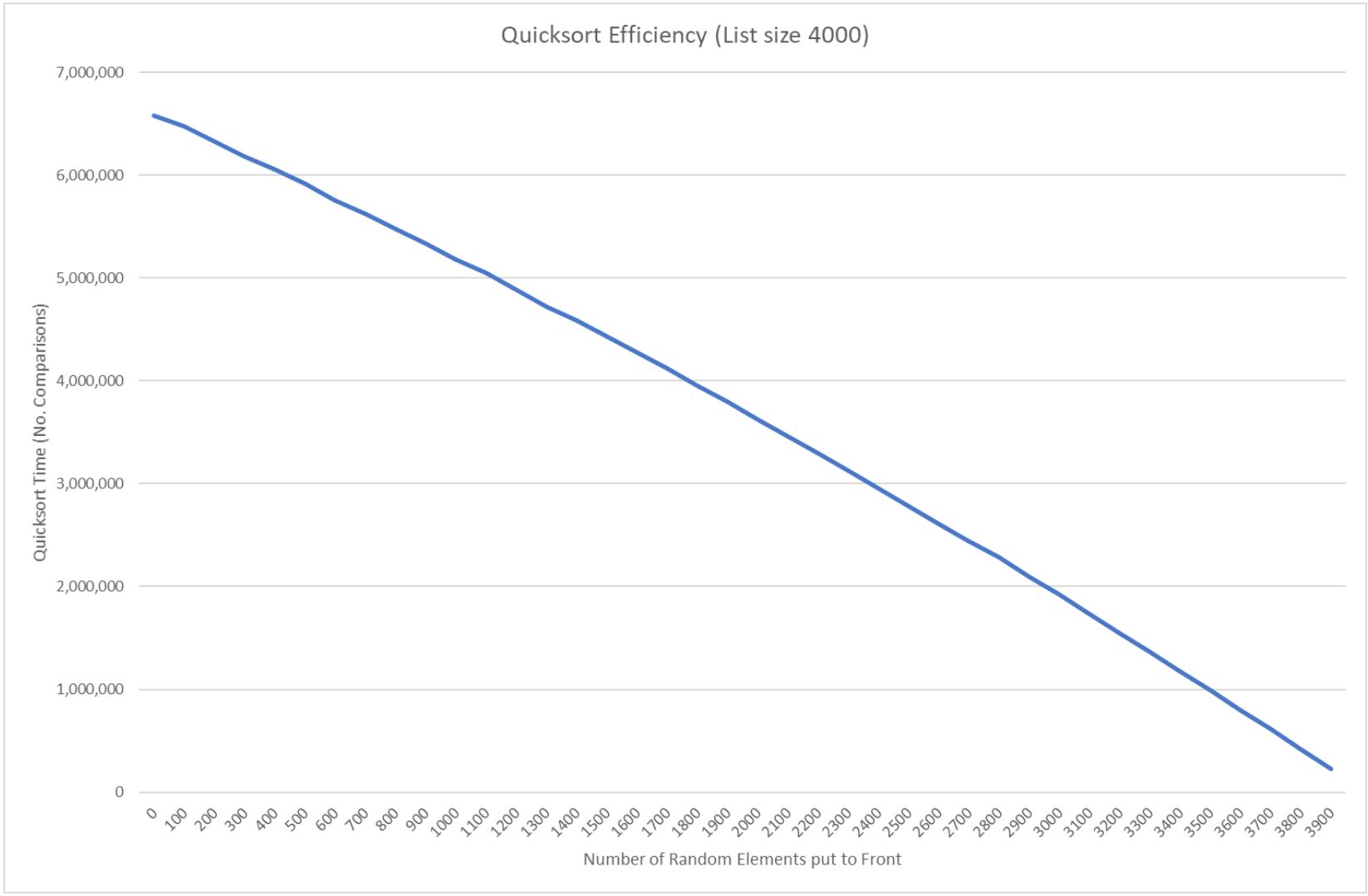
Appendix B

Average Quicksort Efficiency at increments of 10 exchanges.

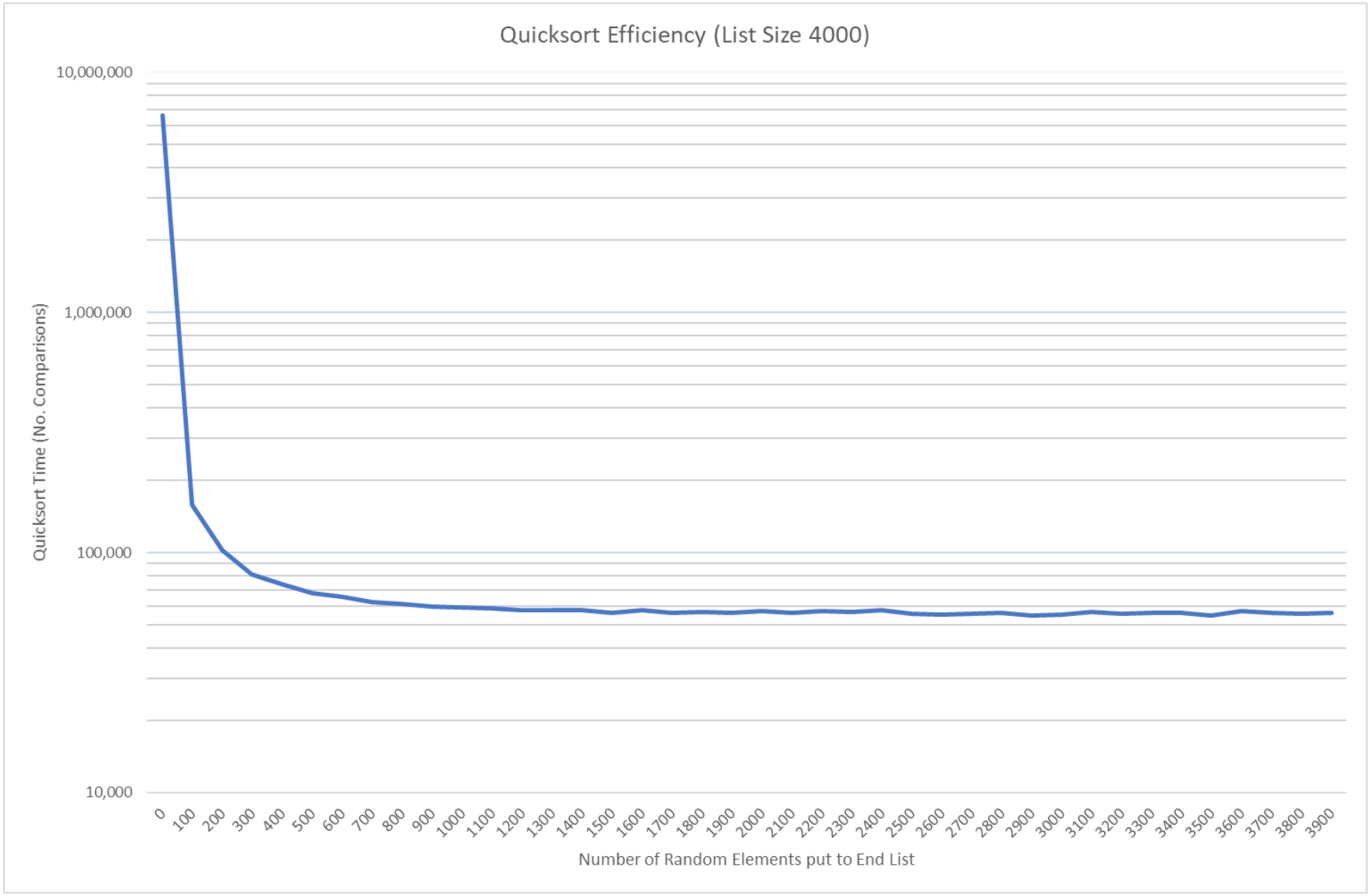


Appendix C

Average Quicksort Efficiency when random elements are taken from the list and put to the head.



Appendix D



Average Quicksort Efficiency when random elements are taken from the list and appended to the tail.

Appendix E

Removal Distance (Add to End)	% Unsortedness (Average)
100 inserts	80.50
200 inserts	78.78
300 inserts	77.40
400 inserts	75.56
500 inserts	73.93
600 inserts	72.16
700 inserts	70.38
800 inserts	68.33
900 inserts	66.83
1000 inserts	64.82
1100 inserts	62.80
1200 inserts	61.13
1300 inserts	59.45
1400 inserts	57.28
1500 inserts	55.35
1600 inserts	53.43
1700 inserts	51.45
1800 inserts	49.43
1900 inserts	47.46
2000 inserts	45.24

Appendix F

Removal Distance (Add to Front)	% Unsortedness (Average)
100 inserts	80.86
200 inserts	79.09
300 inserts	77.20
400 inserts	75.62
500 inserts	73.93
600 inserts	72.19
700 inserts	70.17
800 inserts	68.56
900 inserts	66.73
1000 inserts	64.74
1100 inserts	62.90
1200 inserts	61.03
1300 inserts	59.19
1400 inserts	57.36
1500 inserts	55.24
1600 inserts	53.45
1700 inserts	51.49
1800 inserts	49.54
1900 inserts	47.40
2000 inserts	45.32

Bibliography

https://cs.stackexchange.com/questions/22516/why-is-quicksort-described-as-in-place-if-the-sublists-take-up-quite-a-bit-of

Above is the Pseudocode I used to implement Quicksort.

https://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/ - Source of LIS class.

Curtis, CR and Kim, DJ. (1980). Best Sorting Algorithm for Nearly Sorted Lists. *Communications of the ACM*. 23 (Issue 11), 620-621.

Estivill-Castro, V & Wood, D. (1992). A Survey of Adaptive Sorting Algorithms. *ACM Computing Surveys (CSUR)*. 24 (Issue 4), p. 447.