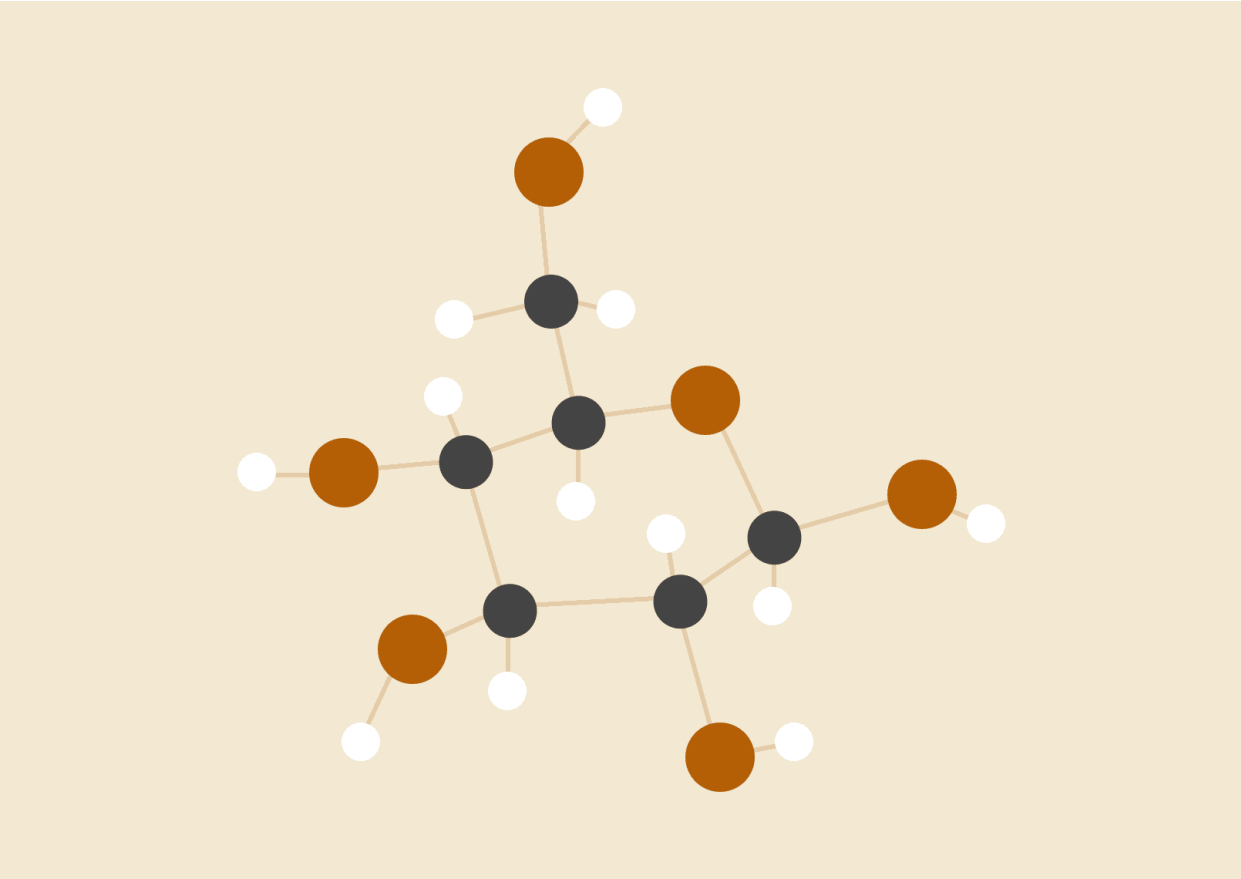
Mergesort

Recursive and Iterative implementation



Blake Hutt

2136113

Hutt0065

Table of Contents

Executive Summary

The report comparisons between the versions of a mergesort implementation based on David Power's pseudo code given in the assignment specification, to produce a Von Neumann iterative and a recursive version of the algorithm. Testing of the code was run in an unix/Linux environment with a maximum sort range of 10,000,000 integers to 10 integers, where as the size decreased the amount of runs increased to a maximum of 1,000,000 runs, such that the total amount of elements remained 10,000,000.

1. Code

The code used in this implementation was C, using the current standard, C11, when compiling. With two different functions:

- mergeSortI - David Power's pseudo code iterative implementation;
- mergeSortR - David Power's pseudo code recursive implementation

These two functions employ the same merge function based on David Power's pseudo code. The differences of these two versions will be discussed below in the differences section.

The merge function takes several parameters, two arrays, a left and right and the size of the array. One array is the to be sorted, named *a* in the code, while the *b* array is a working array to copy values in and out of while they are being sorted, to be passed back into the *a* array once the merge function completes its course in the particular call.

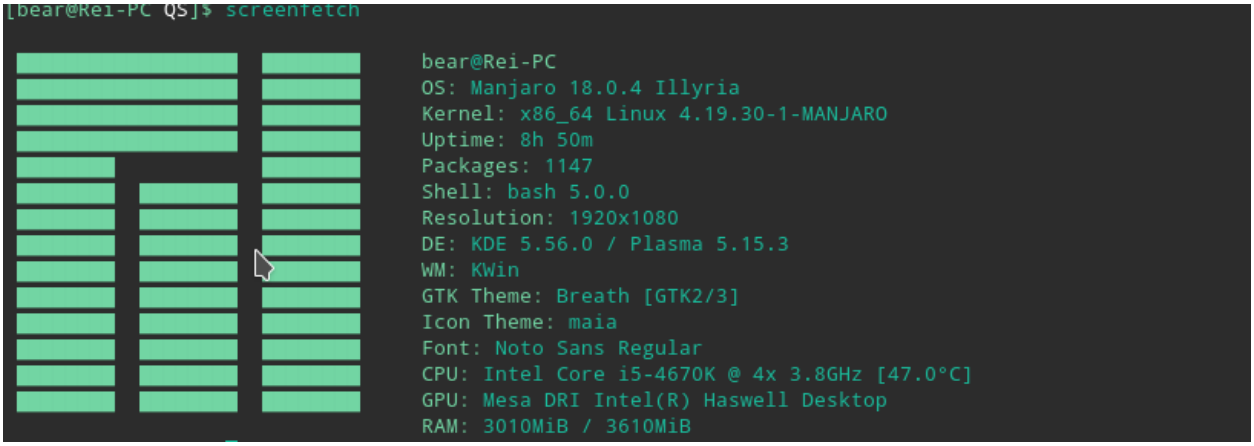
Merge follows the pseudo code given, with an extra variable created called *oLeft* which is used later in the function. Cycling through the array, for each value it compares a left and right pointer, finding the lowest value, copying that value into *b*. It's worth mentioning that the comparison operator of <= is used rather than <, to prevent unnecessary copy operations of the same value, as they are already sorted, leaving them in order.
Depending on which condition breaks the cycling through of the array, it copies the rest of the other split of the array to the end of the working array which is then finally copied to the array to be sorted, using the *oLeft* value to start copying from a point where it won't replace values which are already sorted by previous calls of merge. It then returns the *a* array.

The main function also has a 'checker' to validate if the sorting is working as intended, where it employs an int function, and if it returns a value of -1, it will exit the run and detail which algorithm was the function causing the infraction. This has been extended from the quicksort checker function, to count the amount of duplicates in the runs.

Similar to the quicksort version, each of of the functions, were given their own array to sort, for each and every run. Each array filled with randomised integers(ints), and used the 4 byte variation of ints, so would result in a range from 0 to 2,147,483,647. This could of been changed to unsigned ints to increase the range of possible numbers which would decrease possible repetitions. They also both shared a common work array.

2. Environment

Using the same environment as the quicksort testing to ensure similar results and that it may remove any other issues which would come from using a different testing machine. For completeness, here is the hardware used:



Using the knowledge from the quicksort implementation, the stack size was changed to 120,000kB from get go, and did not result in any segfaults.
The possible use of malloc() and free() was taken into consideration, but decided against as this would change the conditions of the algorithm from using the program stack to the heap, which would increase times slightly more. Giving an unequal future comparison between quicksort and mergesort.

Below is the compiler version used when compiling the code.

```
gcc (GCC) 8.2.1 20181127
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

3. Differences

As above, the iterative and recursive have a different approach in how they sort the array. The recursive function calls itself and follows itself down the left path of the array, continuing until it reaches a size of 1, at that point, it is deemed sorted, then returns up one level and follows the right and follows the right path, this also returns if it is at a size of 1 and returns back one level. It then calls the merge function, sorting the two elements into a sorted pair, copying those to the original array. Repeating as it goes back up the levels, until it starts going down the right path of the original array, repeating the process. Once it has completed the process, it returns the original array fully sorted.

The iterative follows a similar idea, of starting with a size of 1, sub-arrays, it then loops through, finding a mid and a temp right, named *high* in the code, to use as the size of the subarray to be sorted. It then merges them with their ‘neighbour’ into a sorted pair, then quadruplet, etc, depending on the size of the pairs defined in the first loop. This differs from recursive as instead of recursively breaking down the array into singles, pairs, etc. It begins from the bottom and works upward. Creating larger sub-arrays until it reaches the original array size and repeats once more then returns the sorted array.

4. Expectations

4.1 Mergesort Expectations

It is expected that the iterative version of the mergesort implementation would run faster than the recursive versions of the algorithm. As the recursive variation requires to store the function calls in the program stack, with all the relevant data, whilst the iterative does not need to store any such data, except for the initial call. They both require to create copies of themselves when they are being sorted, so the time difference is not expected to be of a great difference, scaling with the size of copying. Expected time comparison between the two a maximum difference of +/- 0.15 seconds, but as mergesort is a stable big-O(nlog(n)), is not expected to differ greatly, regardless of case.

4.2 Comparison of Mergesort and Quicksort

It is generally accepted that mergesort and quicksort are two of the faster algorithms to use with a complexity of Θ(nlog(n)). However, quicksort complexity is based on an average/best case time complexity whereas mergesort is always going to result in an Θ(nlog(n)) behaviour.
This is due to the fact that if a list is already sorted, or reversed sorted, quicksort will increase to an exponent of Θ(n^2), where mergesort will still produce a Θ(nlog(n)) behaviour.
A reason quicksort is often associated with its best case, rather than its worse case, is that it isn't often in a real world scenario where you would be sorting an already sorted list.

So we should expect that they both follow a similar graph and time result which should show a N log(N) growth, or a linear growth if the graph has taken the log of the x-axis. Time expectations would be that mergesort would be on par with the quicksort +/- a couple milliseconds.

5. Results

The general consensus of the of the mergesort results, are in line with the expectations more or less. From the graphs A1 to A2.1, shows that both algortihms followed a nlogn behaviour, while they were increasing, they are not increasing in such a fashion that is not expected. This is confirmed by the linear graphs where the log of the x-axis has been taken, showing that indeed the algorithm is running at an Θ(nlog(n)) complexity.

With the iterative version of mergesort outperforming the recursive from a minumum of 0.03 to a max of 0.09 seconds in all tests, with a .07 second difference in the largest singular test of 10,000,000 elements. This netting the iterative a ~3.6% performance boost, which could be seen as negligible. These close runtimes of the iterative and recursive are shown in graphs A3 to A4.1.

Using the results from the quicksort review, excluding Hoare's implementation, compared both the total runtimes of the recursive and interative versions of sorts. Tabling and graphing the algorithms for comparisons immeadiately showed a large difference between the two, with quicksort seeing an all-round lower runtime. While it was in-line with the expectations, the actuality of quicksort being a maximum of 0.5 seconds faster in the recursive comparison and 0.4 seconds faster in the iterative versions at the max 10,000,000 size, is none the less shocking. After considering the possibilities, the mergesort is taking longer due to having to copy the array to be sorted and copying back once it is sorted, creating a bit of excess time. Similarly, we created the quicksort as an in-situ sort, where it does not require an extra bit of memory, this could leave more space for it to be stored in a faster storage locations of the CPU cache levels 1-3, while mergesort might have been moved to and from RAM more frequently due to the increased size of creating a extra array to be used in operations.

However, it is seen that both algorithms are following the expected Θ(nlog(n)) result and the linear growth when the log of the x-axis has been taken.

6. Conclusion

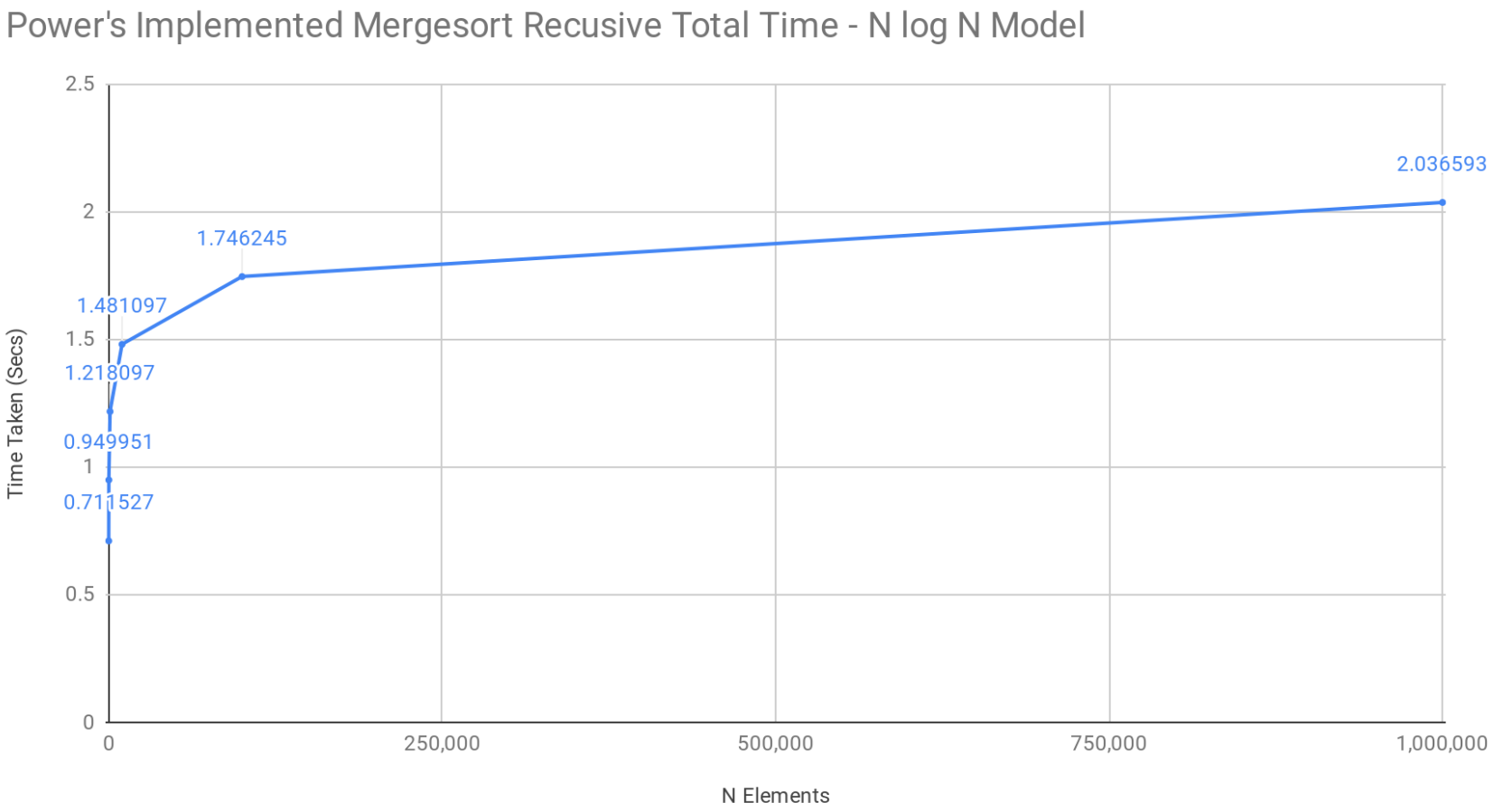
From what was expected from the two variations of mergesort algorithm, the iterative performed better than the recursive, however, the performance benefit could of been seen as negligible at this size of sorting, it may see better at even larger sizes where it would scale. As touched on, in the comparison of quicksort vs mergesort, it may also see improvement where the CPU and RAM speeds are increased and having a larger CPU cache would see improvement in the runtime. This would also see improvement in quicksort runtime too.

The comparison of quicksort and mergesort also provided some more insight into the two algorithms and how hardware can also play a role in the runtime of algorithms. But it is important to remember that they both follow the $\Theta(n\log(n))$ expectation as seen in the graphs.

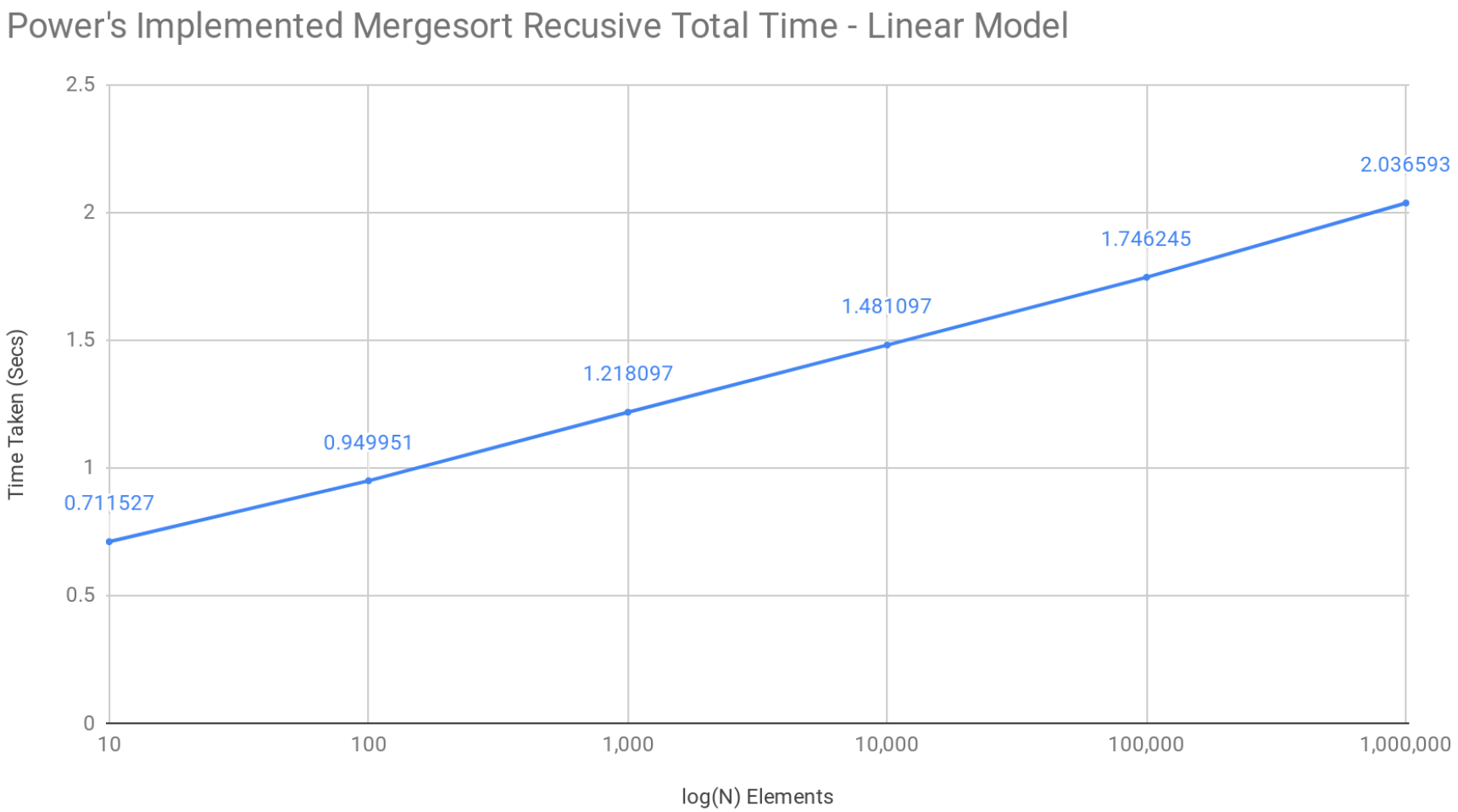
For possible improvements, as mentioned in section 2, would be looking into implementing malloc calls for the arrays, giving it more portability to other environments where changing the stack size is prevented, although stack variables are faster than using the heap. Another possible improvement, would be removing the work array from main, and creating a unique temporary work array in the merge function, this would allow for thread usage with the child threads working the different paths to return to the partent thread to perform the merges, with a bit more added complexity.

Appendices

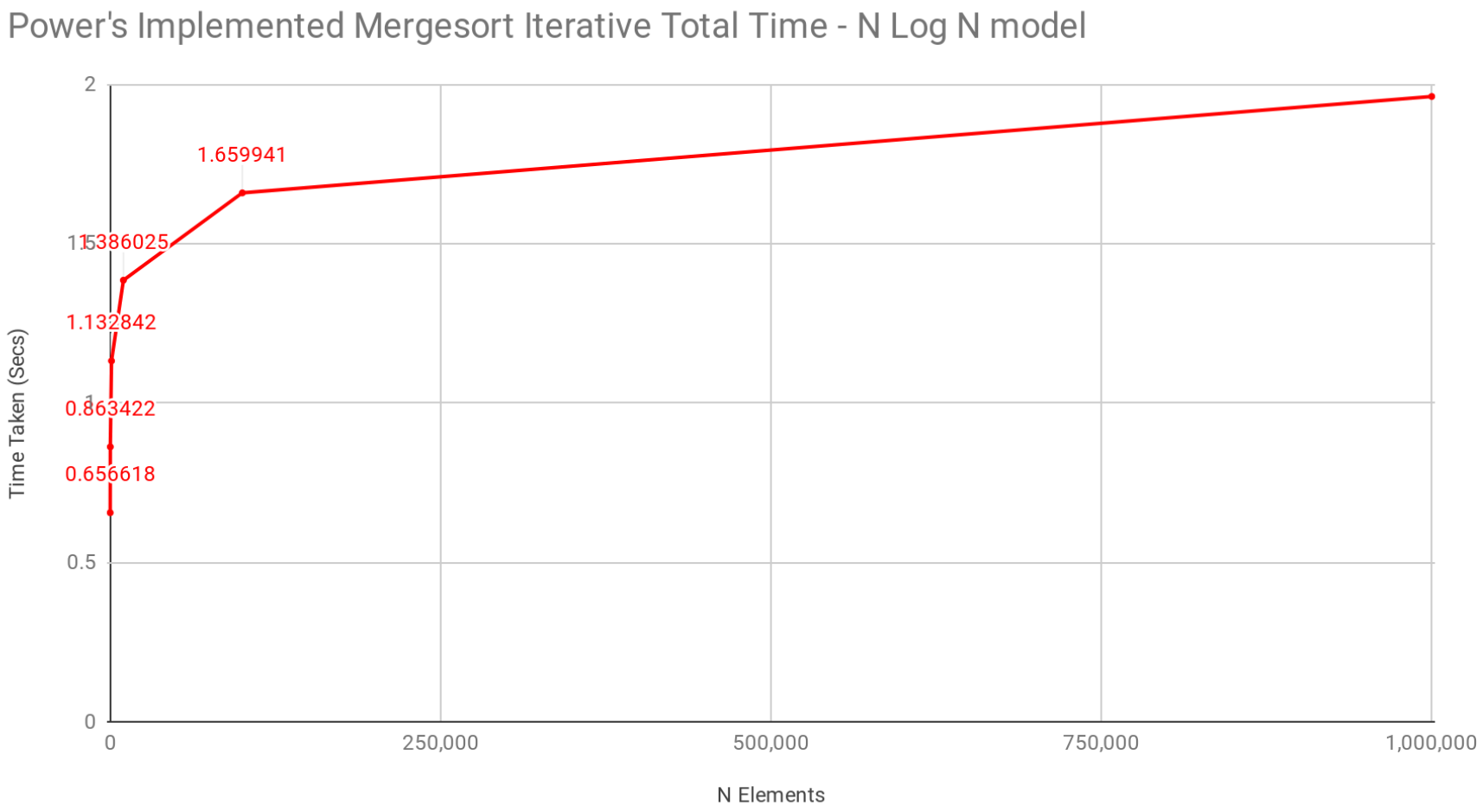
A1: Recursive with n array elements



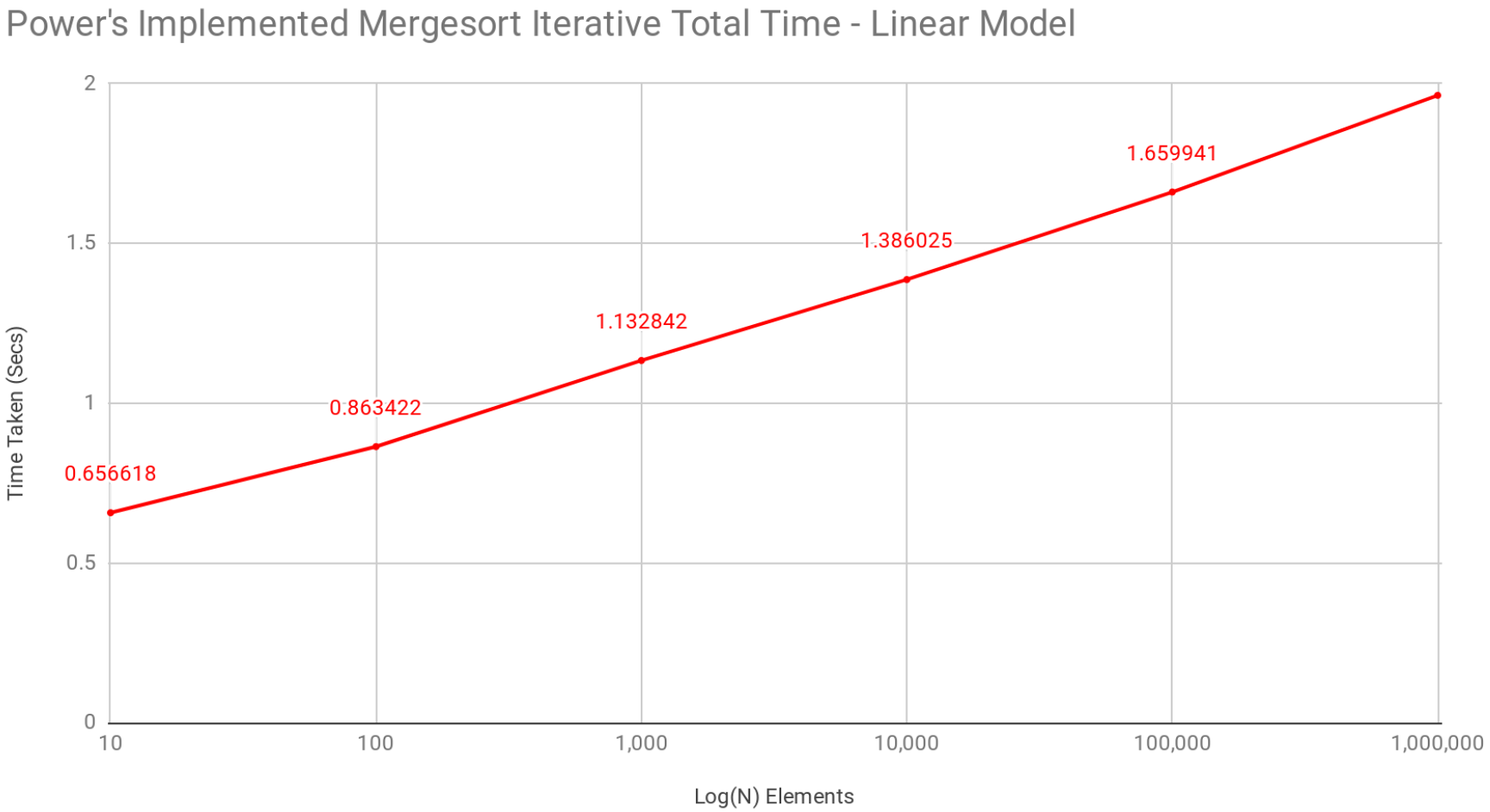
A1.1: Recursive with log n array elements



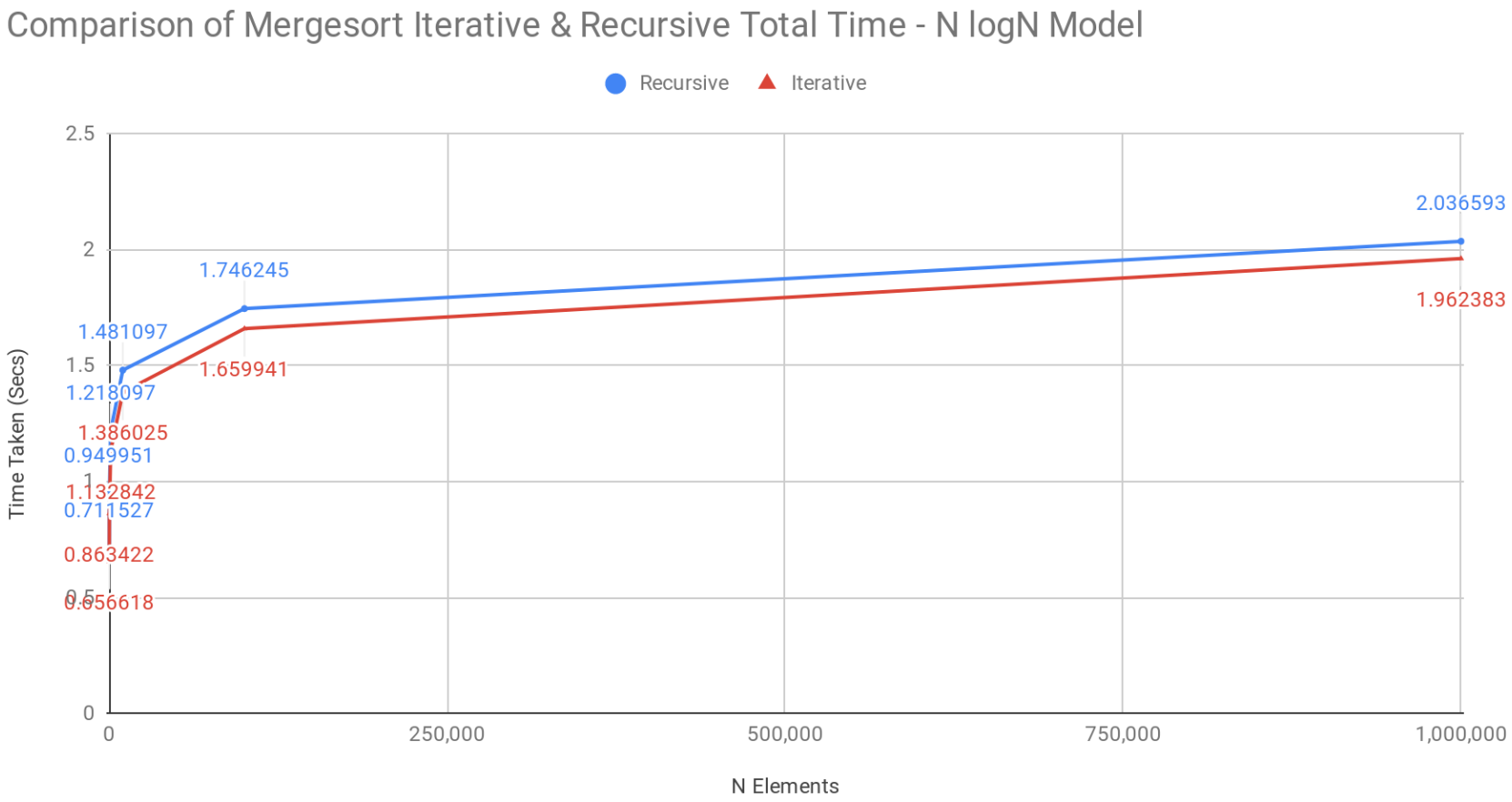
A2: Iterative with n array elements



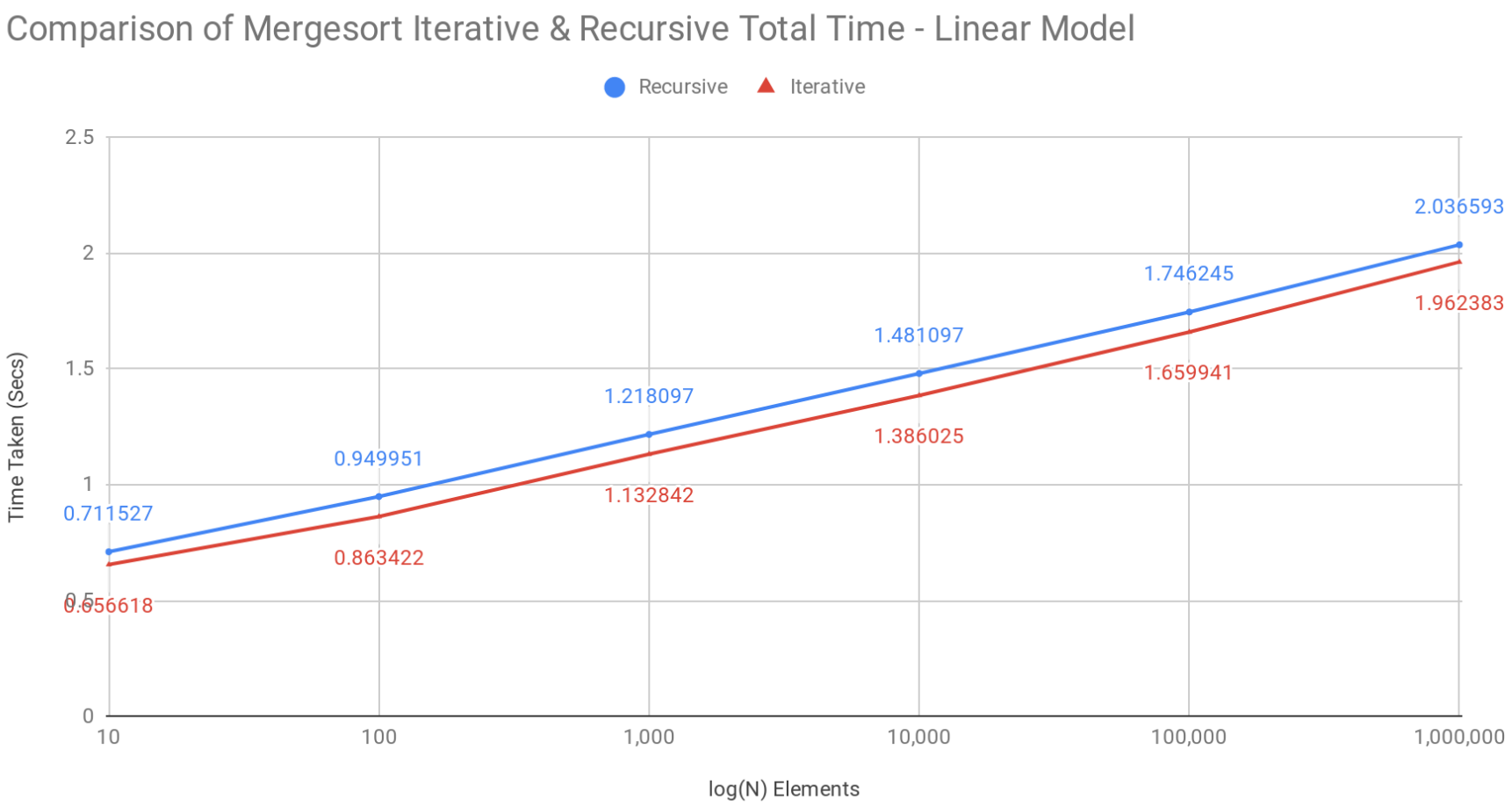
A2.1: Iterative with log n array elements



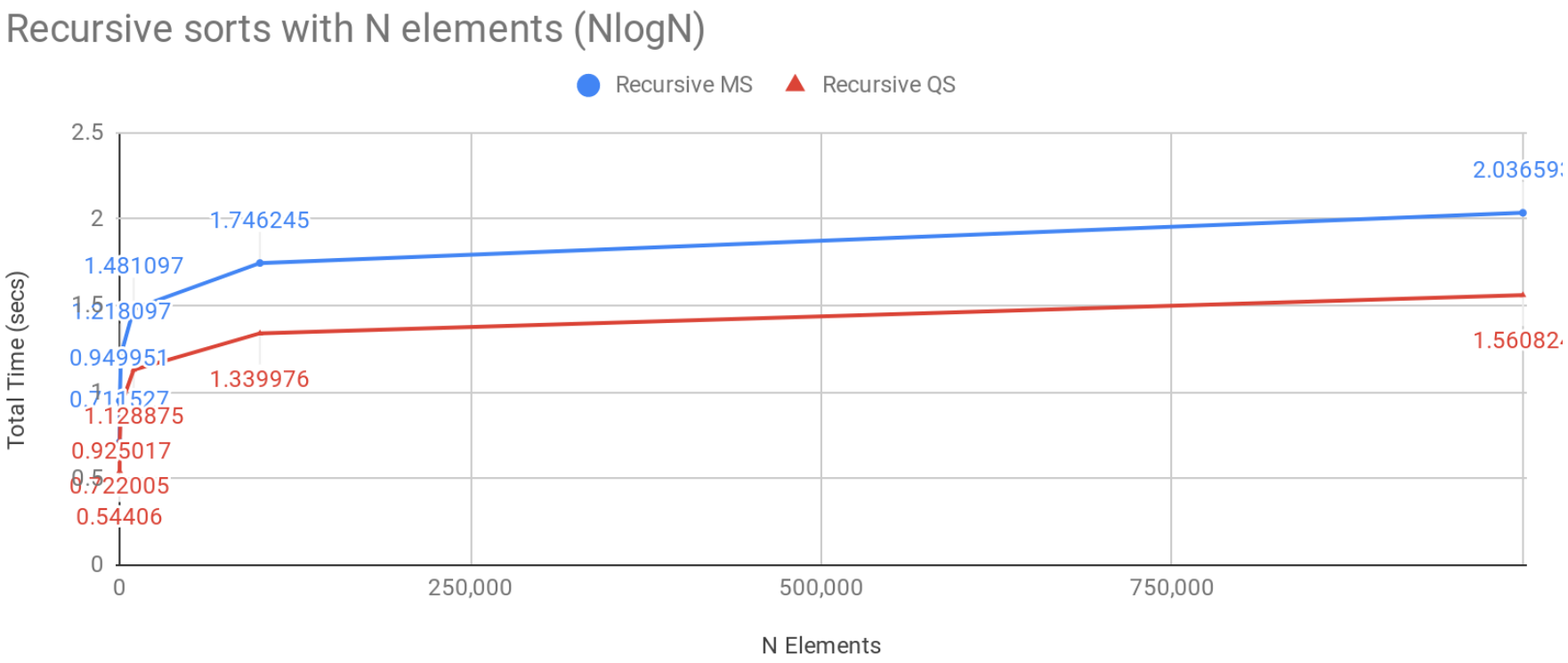
A3: Time comparison of n array elements



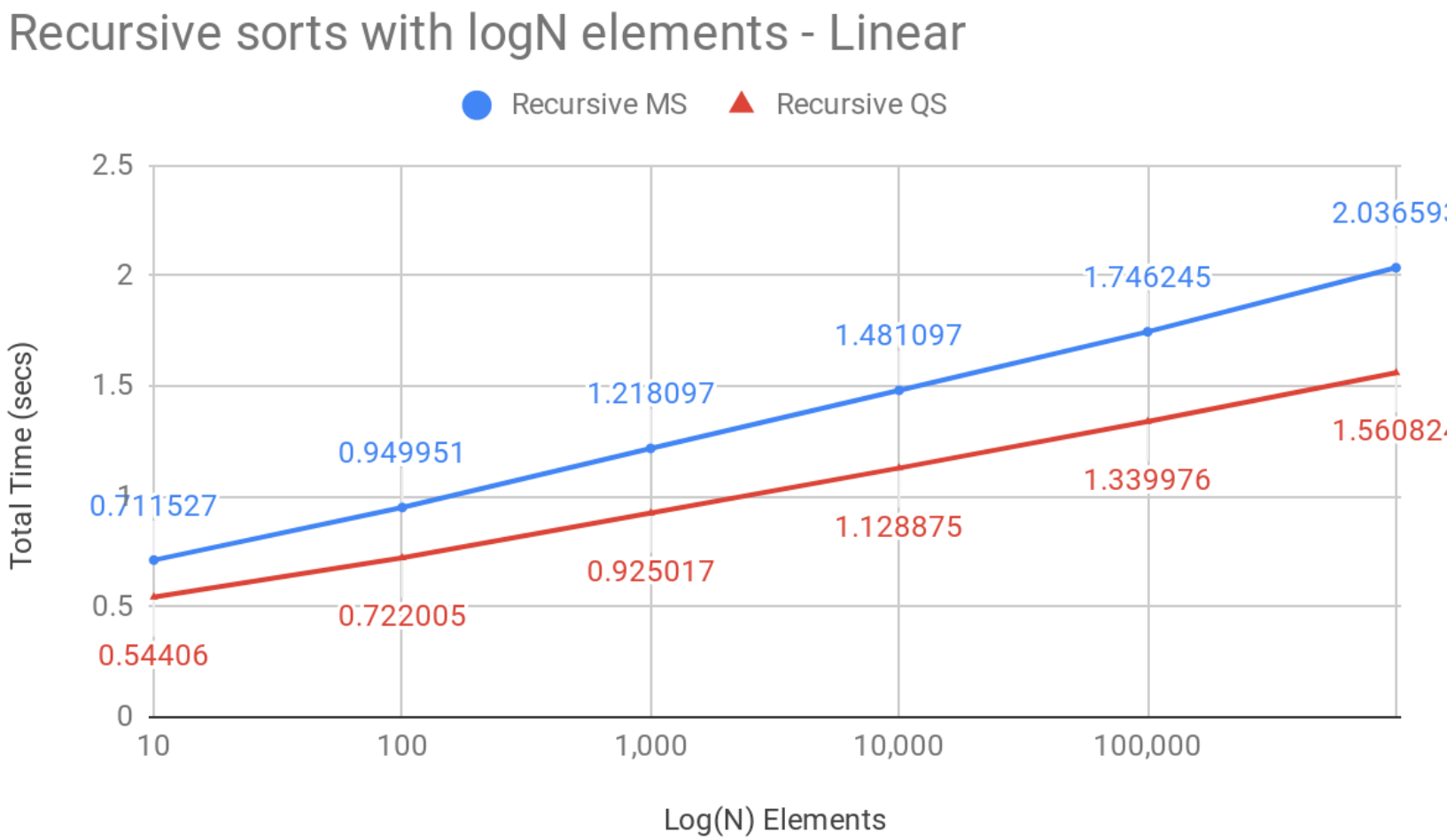
A3.1: Time comparison with log n array elements



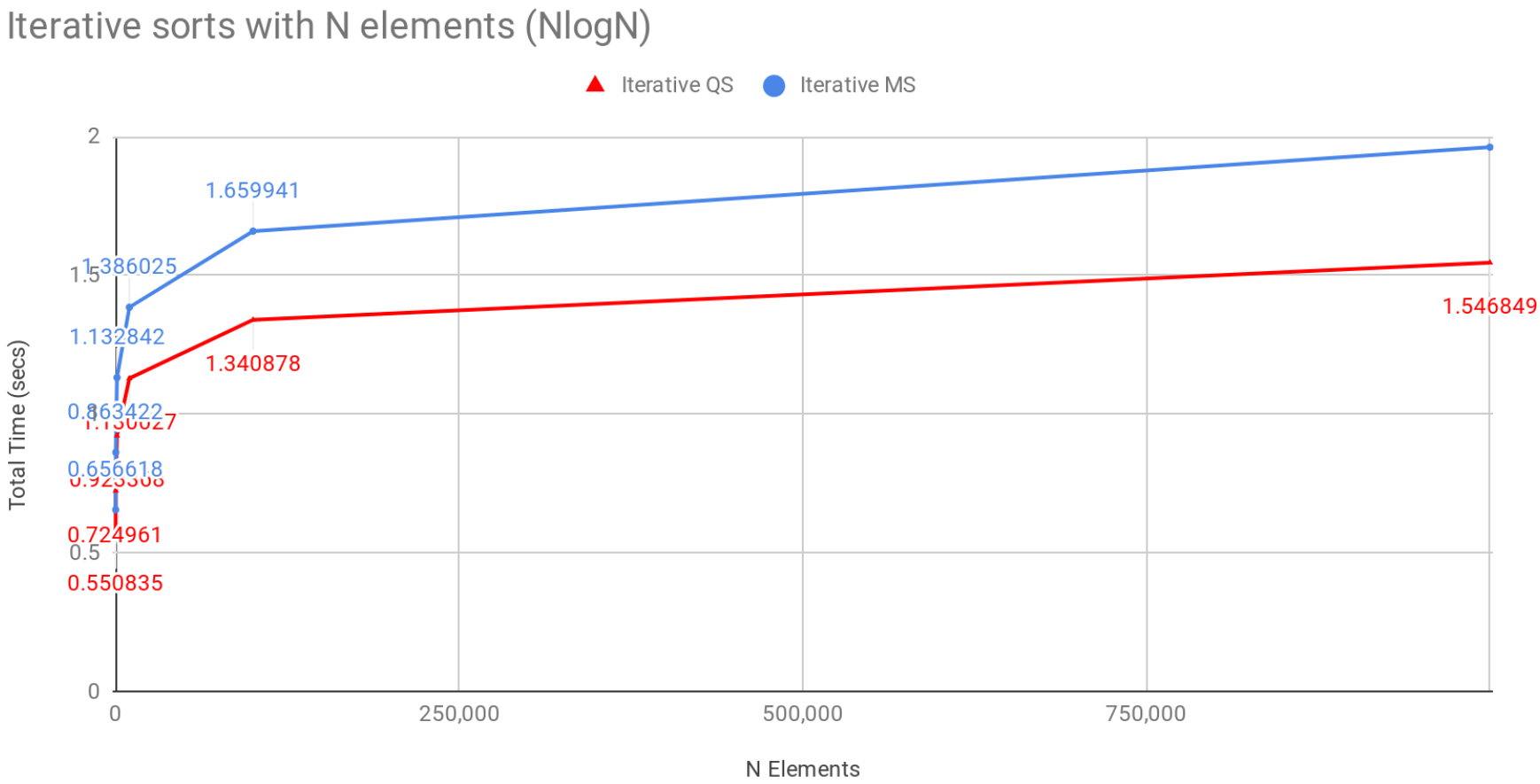
A4: Time comparison of recursive quick and merge sorts with n array elements



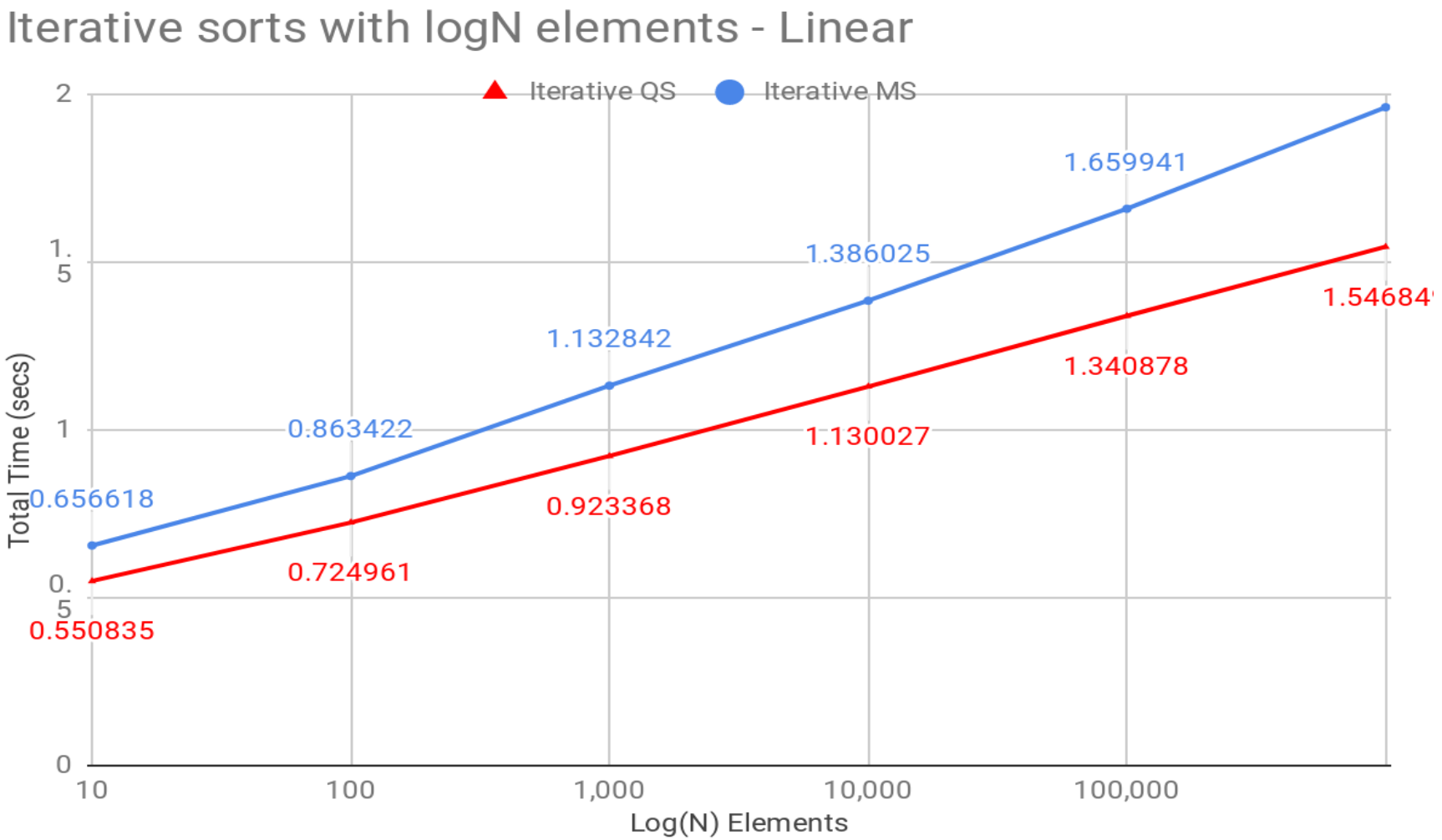
A4.1: Time comparison of recursive quick and merge sorts with log n array elements



A5: Comparison of Power's recursive quick and merge sorts with n array elements



A5.1: Comparison of recursive Power's quick and merge sorts with log n array elements



A6: Tabulated Data - Total Time in seconds

A6.1 Mergesort total time and amount of duplicates

No. of Elements	Recursive	Iterative	Rec. Dupes	Iter. Dupes
10,000,000	2.036593	1.962383	23110	23235
1,000,000	1.746245	1.659941	2415	2310
100,000	1.481097	1.386025	227	235
10,000	1.218097	1.132842	16	24
1,000	0.949951	0.863422	2	4
100	0.711527	0.656618	0	0
10	0.725273	0.692352	0	0

A6.2 Mergesort and Quicksort total time comparison

No. Of Elements	Recursive QS	Recursive MS
10,000,000	1.560824	2.036593
1,000,000	1.339976	1.746245
100,000	1.128875	1.481097
10,000	0.925017	1.218097
1,000	0.722005	0.949951
100	0.54406	0.711527
10	0.658074	0.725273
No. Of Elements	**Iterative QS**	**Iterative MS**
10,000,000	1.546849	1.962383
1,000,000	1.340878	1.659941
100,000	1.130027	1.386025
10,000	0.923368	1.132842
1,000	0.724961	0.863422
100	0.550835	0.656618
10	0.722427	0.692352

A7: Raw Data

See included .zip folder for raw/output data.

A8: Individual Quicksort graphs

See quicksort_A review report.