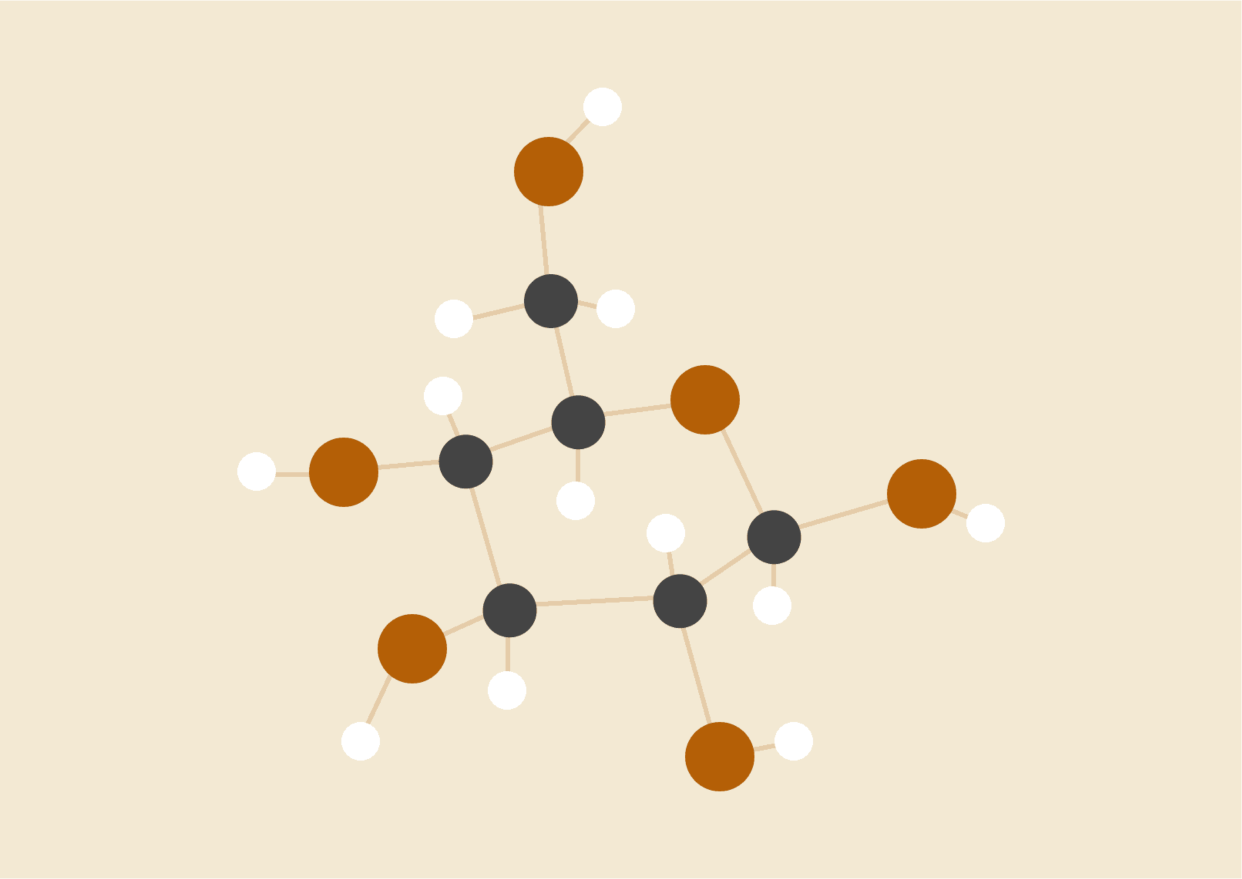
Quicksort

Recursive, Iterative and Hoare's implementation



Blake Hutt

2136113

Hutt0065

Executive Summary

The report comparisons between the versions of a quicksort implementation based on David Power’s pseudo code given in the assignment specification, an iterative and a recursive version. Both of these will be compared against Hoare’s pseudo code implementation, which can be found on wikipedia and other sources. 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. This was repeated thrice, to produce an average number for all the total runs. This showed that both the recursive and Hoare’s implementation produce a linear graph in a log n environment, whilst the iterative result was unexpected.

Code

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

- iterativeQS - David Power's pseudo code iterative implementation
- recursiveQS - David Power's pseudo code recursive implementation
- hquickSort - Hoare's recursive quicksort

The first two functions employ the same partition function based on David Power's pseudo code, whilst the latter uses Hoare's partitioning scheme. The two partition schemes follow a similar layout, but have a subtle described in below in the report.

The Power's implementation of both recursive and iterative follow the same start up and use similar methodologies of using the smaller partition first, this is to minimise stack size and improve memory usage. The iterative version stores the larger partition first, however, then proceeds to add the smaller partition this is to achieve a First-In, Last-Out or FILO, then popping the smaller partition to sort and beginning the process again. Where the recursive once it reaches the first call, will store the current call in its stack and start a 'sub-process' and begin the function again repeating until it reaches a point where the left boundary is no longer less than the right boundary, at this point it has reached a one element array.

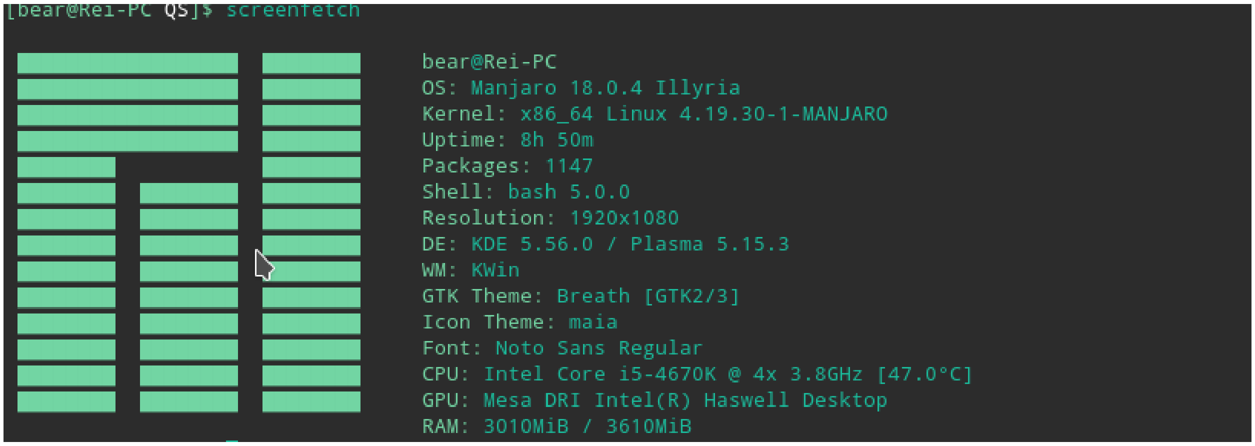
Both recursive implementations will be discussed below in the differences.

The main function also has a 'checker' to validate if the sorting is working as intended, where it employs a boolean function, and if it is false, it will exit the run and detail which algorithm was the function causing the infraction. This could also be extended to check for the amount of duplicates in the arrays too, if desired.

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.

Environment and issues

The code testing was done on a Linux environment described below:



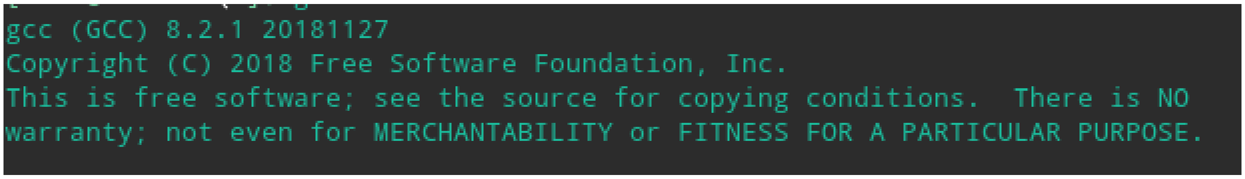
```
[bear@Rei-PC QS]$ screenfetch

                              bear@Rei-PC
                              OS: Manjaro 18.0.4 Illyria
                              Kernel: x86_64 Linux 4.19.30-1-MANJARO
                              Uptime: 8h 50m
                              Packages: 1147
                              Shell: bash 5.0.0
                              Resolution: 1920x1080
                              DE: KDE 5.56.0 / Plasma 5.15.3
                              WM: KWin
                              GTK Theme: Breath [GTK2/3]
                              Icon Theme: maia
                              Font: Noto Sans Regular
                              CPU: Intel Core i5-4670K @ 4x 3.8GHz [47.0°C]
                              GPU: Mesa DRI Intel(R) Haswell Desktop
                              RAM: 3010MiB / 3610MiB
```

It may also be worthy of mentioning the CPU uses a Little Endian byte ordering.
As can be seen, the specification of the hardware is neither outstanding nor poor. With possibly the memory being the only shortfall.

While trying to run the code, some issues arose, those being related to segfaults. With quick debugging, it came to notice that it crashed as soon as the program started filling the array with ints when the size was above 650,000 with crashing obviously occuring at 10,000,000. From there, it took a while to realise, but however it was soon understood, that the arrays where exceeding the stack allotment given by the computer for executing processes, which is soft defaulted to 8192kB, or 8MB. Thus needed to change the possible stack size for processes, calculating that a 10,000,000 int array would require a 40MB stack, with the 4 byte ints, the process was given 130MB stack, suitable for each array, with a little remainder. Consideration was made to change the code to use the heap allotment with the malloc() call, however this made changes in how the sorting happened, so was discarded to be done in own time for further expansion and understanding.

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.
```

Differences

As mentioned above, the partitioning schemes employed by the functions differs slightly, where they both increment and decrement the left and right pointer, respectively, until they find a value which is higher/lower than the pivot and swapping values, or they cross positions with each other. However, this is where they differ, in Hoare's partition scheme has the value of the right pointer returned as the new pivot to be used again, whilst the former partition scheme only returns the right pointer if the value of the pointer is lower than the pivot, else it returns the original pivot declared at the start of the function.

Similarly, the recursive and Hoare's implementation follow a similar structure, where they call themselves, unless the left bound is greater than the right bound, where it would be sorted or an array the size of 1, then it calls the left partition first, which is not always the smallest partition. The recursive implementation follows with the left and right bounds, however, it has another check where if the left bound is lower than the pivot it would run that recursive call first, because the left side is the smaller partition, else the right is the smaller and will run that recursive call.

Expectations

It would be expected that the iterative version of the quicksort implementation would run faster than both the recursive versions of the algorithm, this is due to the fact that it implements its own stack-like memory, of a predefined size, of a Log N of the original array, with the integers to be sorted, rather than the recursive versions using an implicit stack; with the function call being stored on the stack to be pushed and popped off.

Between the two recursive functions, it would be expected that the implementation of Power’s pseudo code would run faster, as described above in the differences, the check to see which partition is the smaller, to use less memory and space in the implicit stack before running the larger partition first, as it calls the small partition first unlike Hoare’s implementation, which will always run the left partition first regardless of size.

Results

The general consensus of the results, are in line with the expectations more or less. With both pseudo code implementations given by Power's performing quicker in all runs and on average, bar the iterative version, falling short in the 10 sized arrays. A probable cause of this may be due to the constant time taken to create two array stacks of log N size, every run, inflating the time taken, which seems to follow as it begins to become faster than the other versions at higher array sizes and less runs.

It is interesting to note, that all versions of quickSort completed on average the 100 sized arrays quicker than the 10 sized arrays. Especially, interesting that the iterative implementation sees a performance increase of 0.2 seconds, seeing a -23% time taken to complete. The implemented recursive version sees a near similar decrease in time taken with a -17%.

What is surprising from the results and the averages, is that the implemented recursive is performs ever so slightly quicker than the implemented iterative version, but it is negligible. It is thought of that iterative algorithms tend to fair better than recursive alternatives, in speed and memory. If its an implementation issue, which it may of been, would see as to why it performs slightly slower. Measuring the memory consumption of the application, was not taken, thus no comparison can be made on that front.

From the graphs, it can be seen however, that all the average times show an increasing linear line when it is modelled against the log elements and when it is modelled against N elements, show a concave down curve, which could possibly be seen easier if a greater amount of memory was given to subsidise the array memory issues, or look into implementing a heap version of the algorithm with malloc().

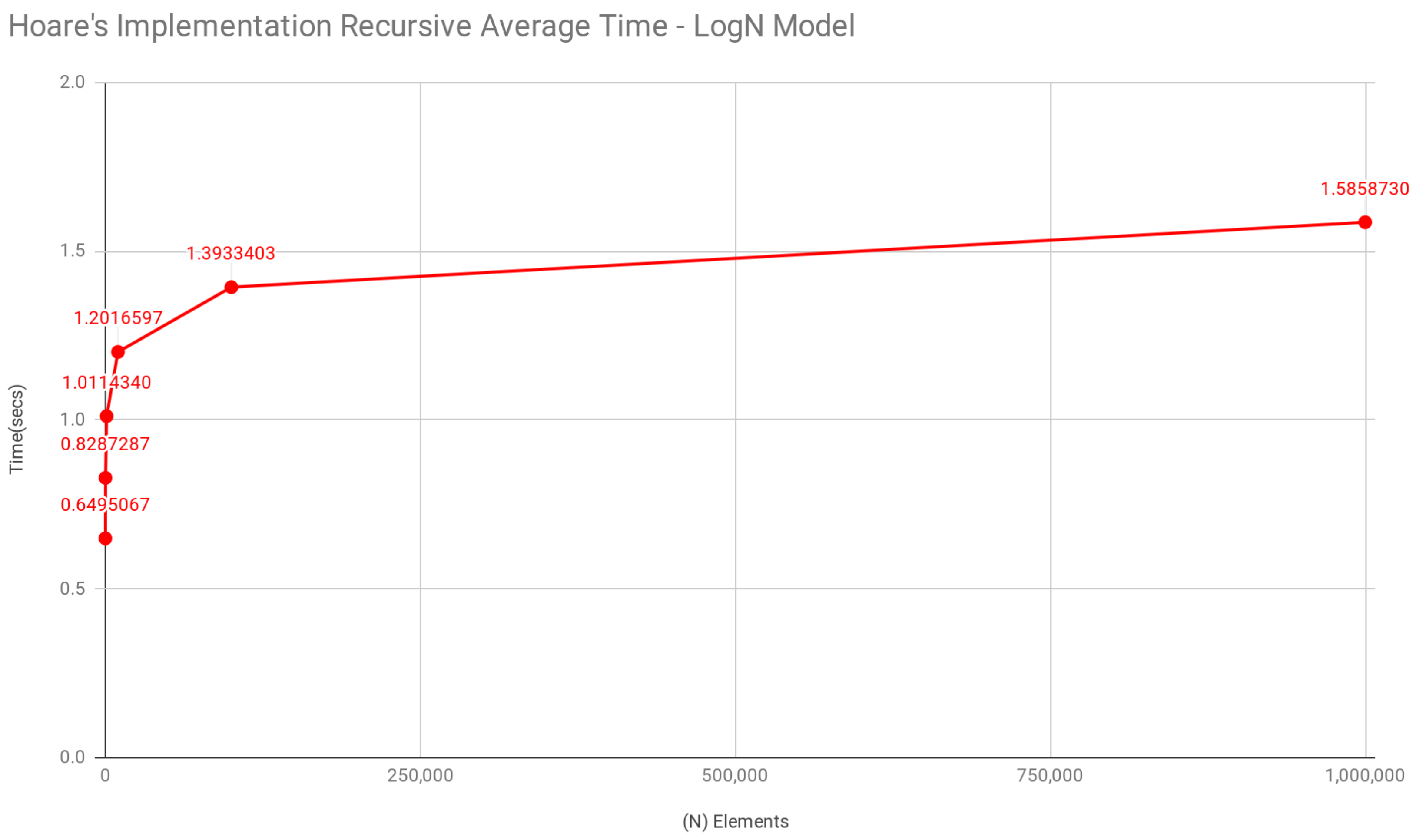
Conclusion

From what was expected from the three different versions of the quicksort algorithm, two following the pseudo code given and Hoare’s pseudo code, that the iterative version would perform better than both recursive variations, which was however incorrect, by a small margin, performing slower than the recursive version given by Power’s, but was still faster than Hoare’s implementation.

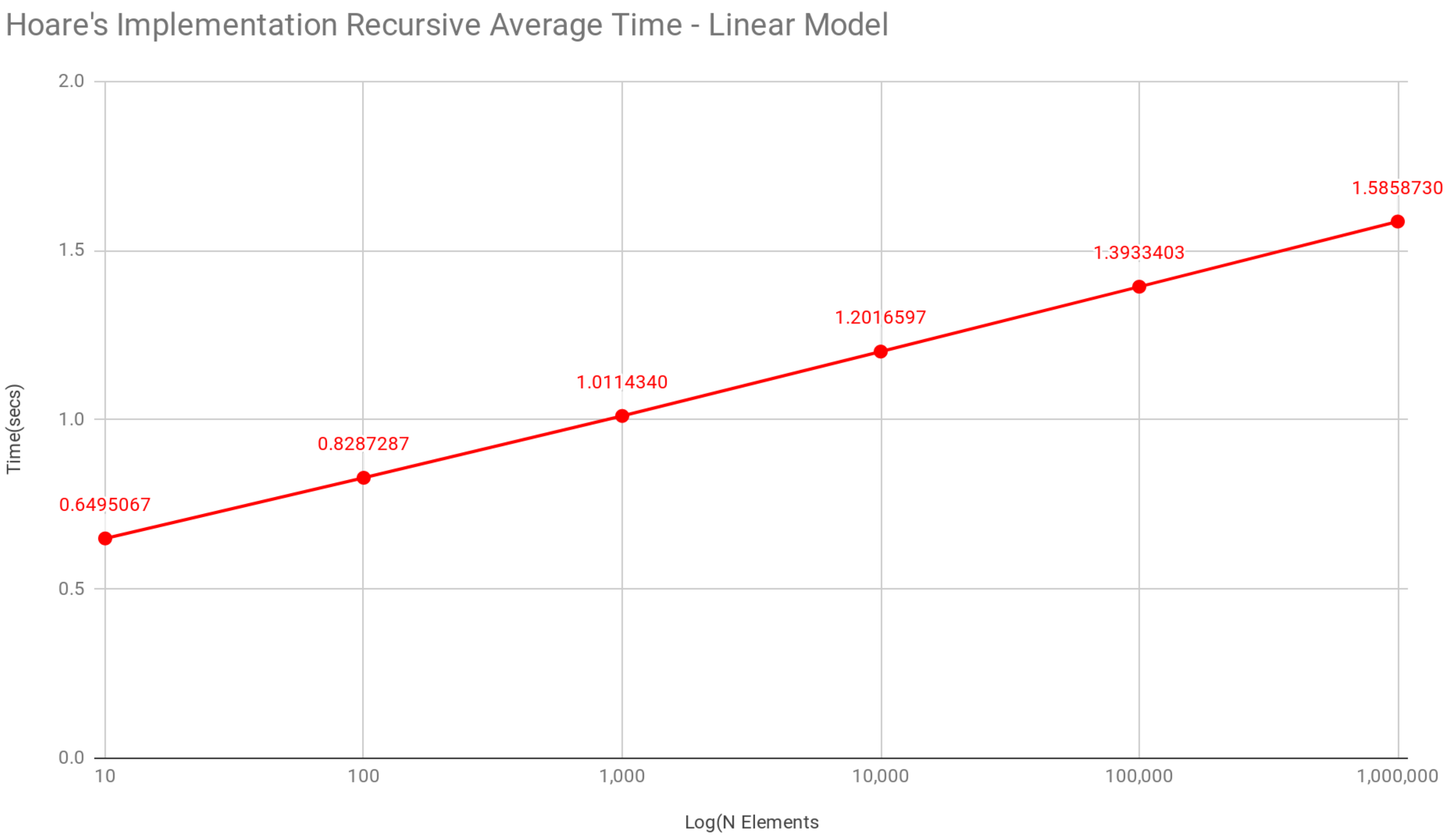
For possible improvements, would be looking into implementing malloc calls for the arrays, giving it more portability to other environments where changing stack size is prevented or obfuscated without privileges, ie Windows. This may see some improvement in speeds too for all versions, although stack variables are faster than using the heap.
Furthermore, implementing a different pivot choice for the iterative and recursive versions, as they use the leftmost element, while Hoare’s does not. This would minimise the chance for poor cases where the arrays are already sorted, but not wholly prevent them. Similarly, keeping track of duplicates and even them out between the partitions to prevent excessive increase in operations.

Appendices

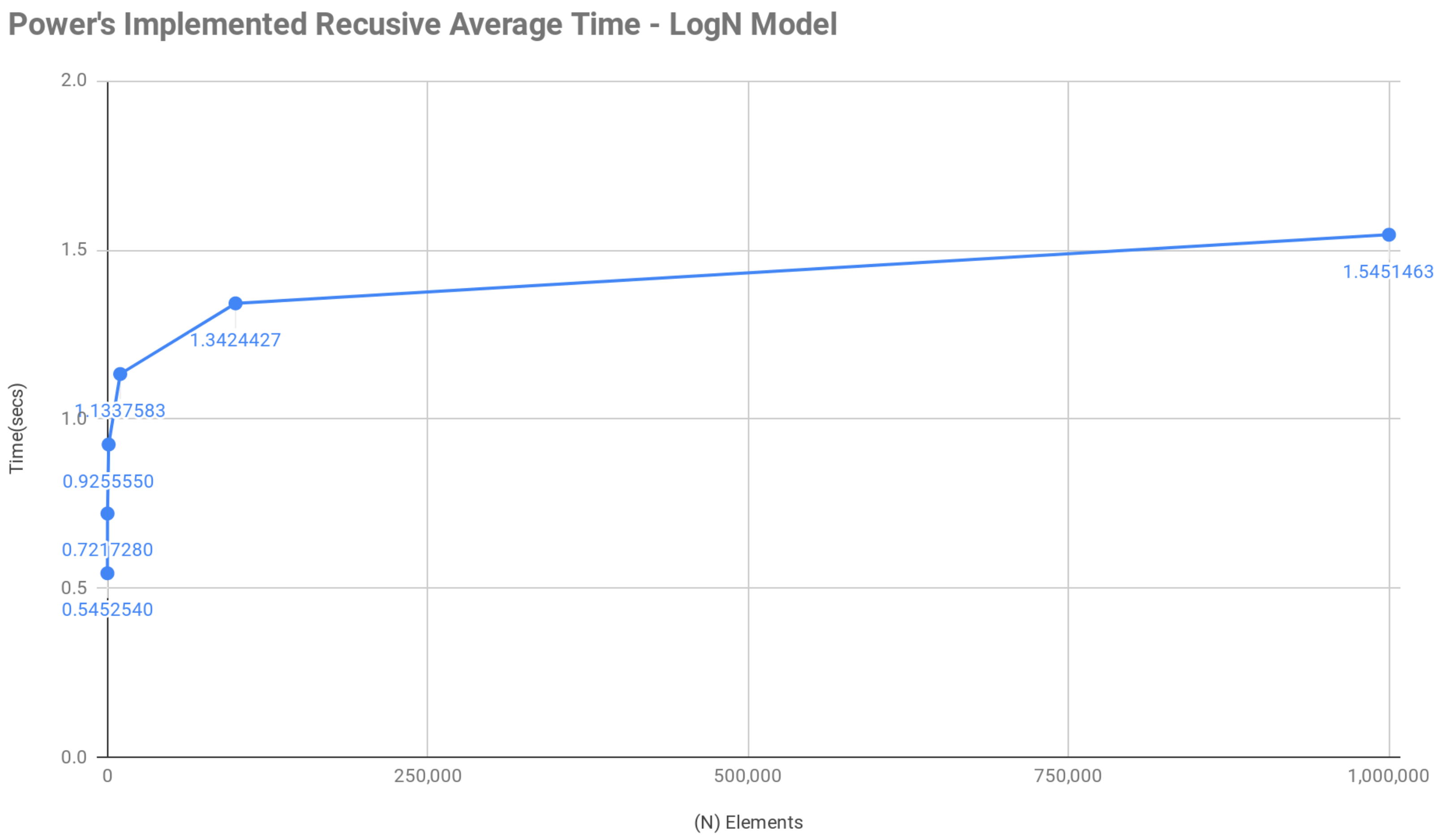
A1: Hoare’s Recursive with n array elements



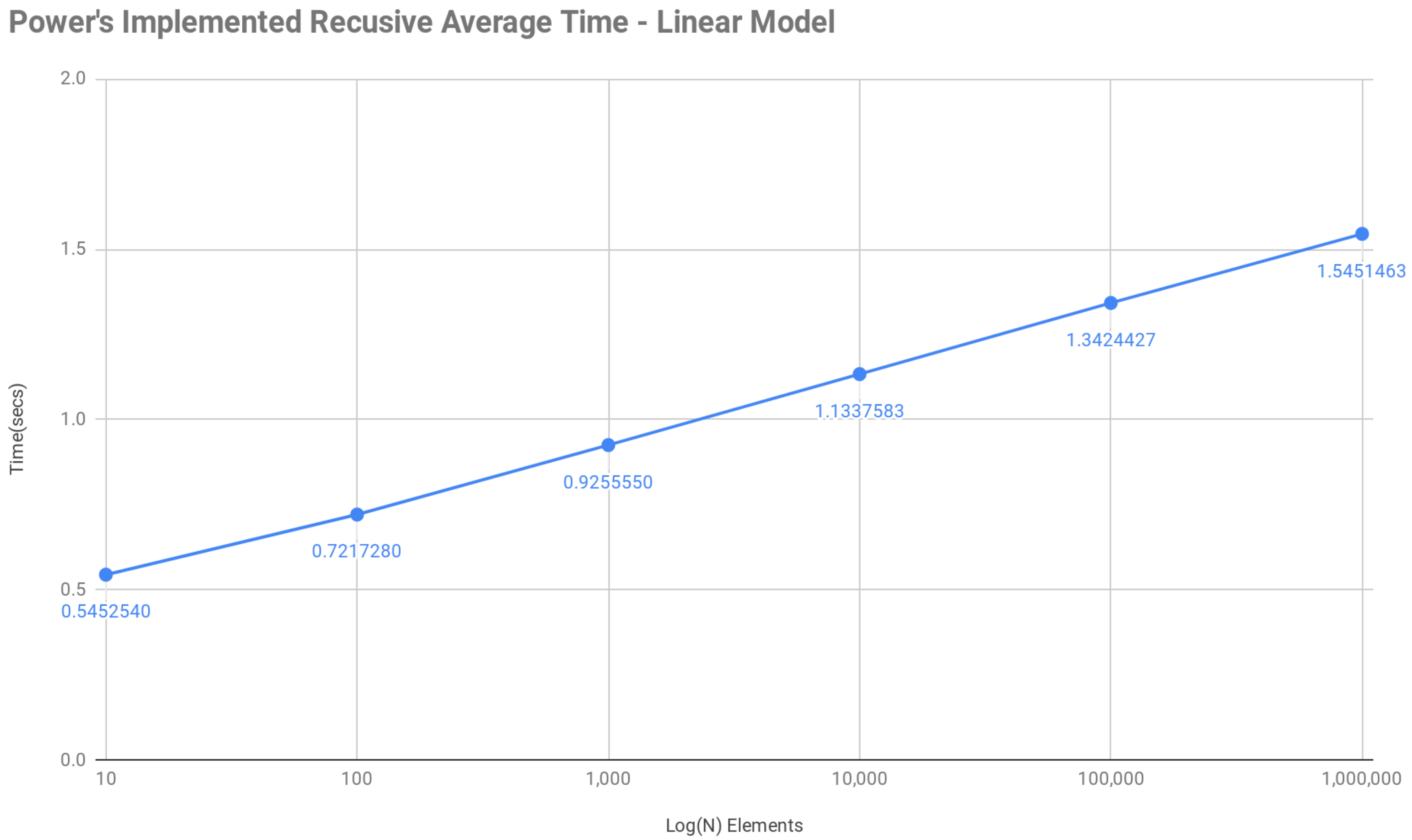
A1.1: Hoare’s Recursive with log n array elements



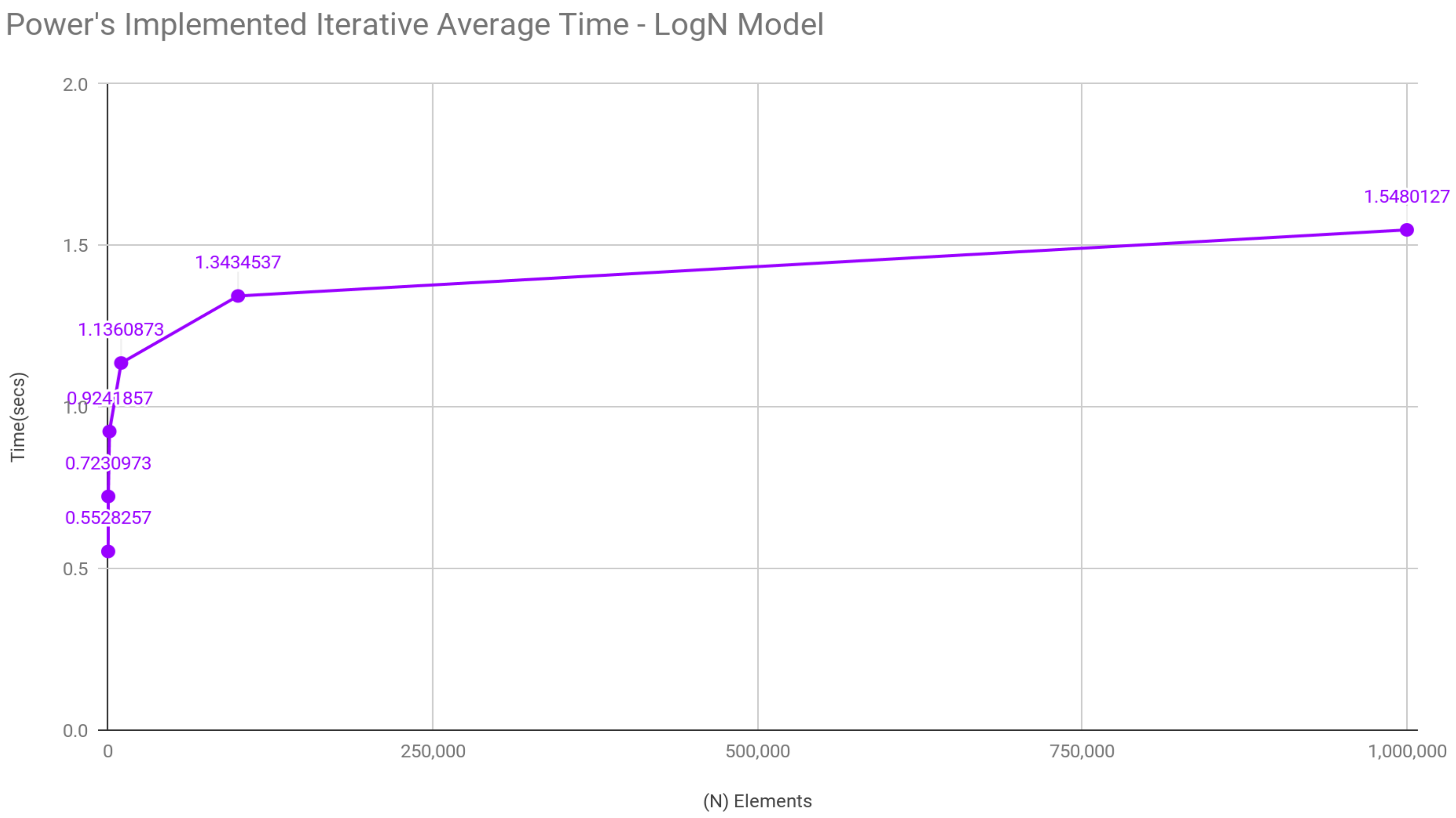
A2: Recursive with n array elements



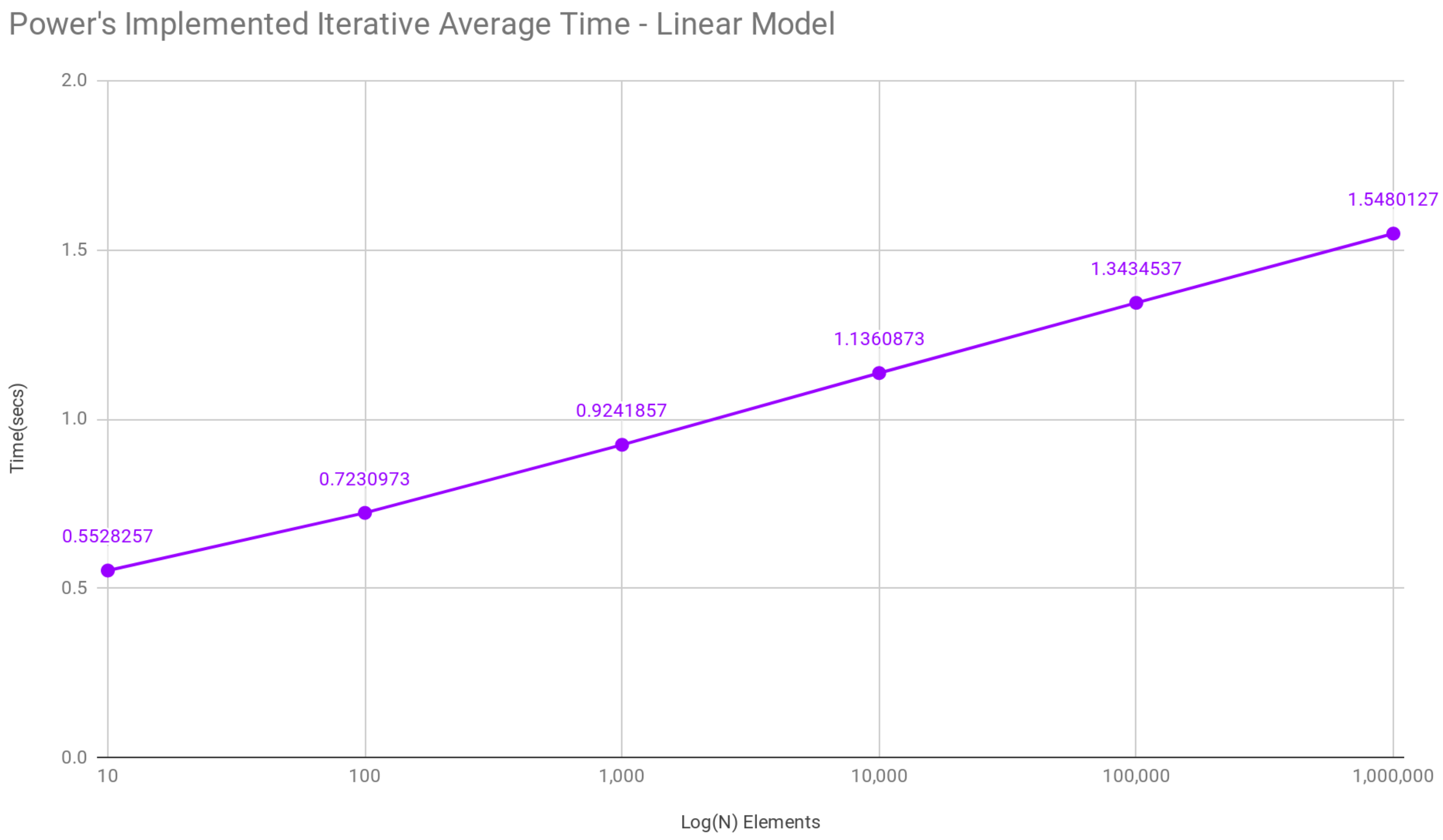
A2.1: Recursive with log n array elements



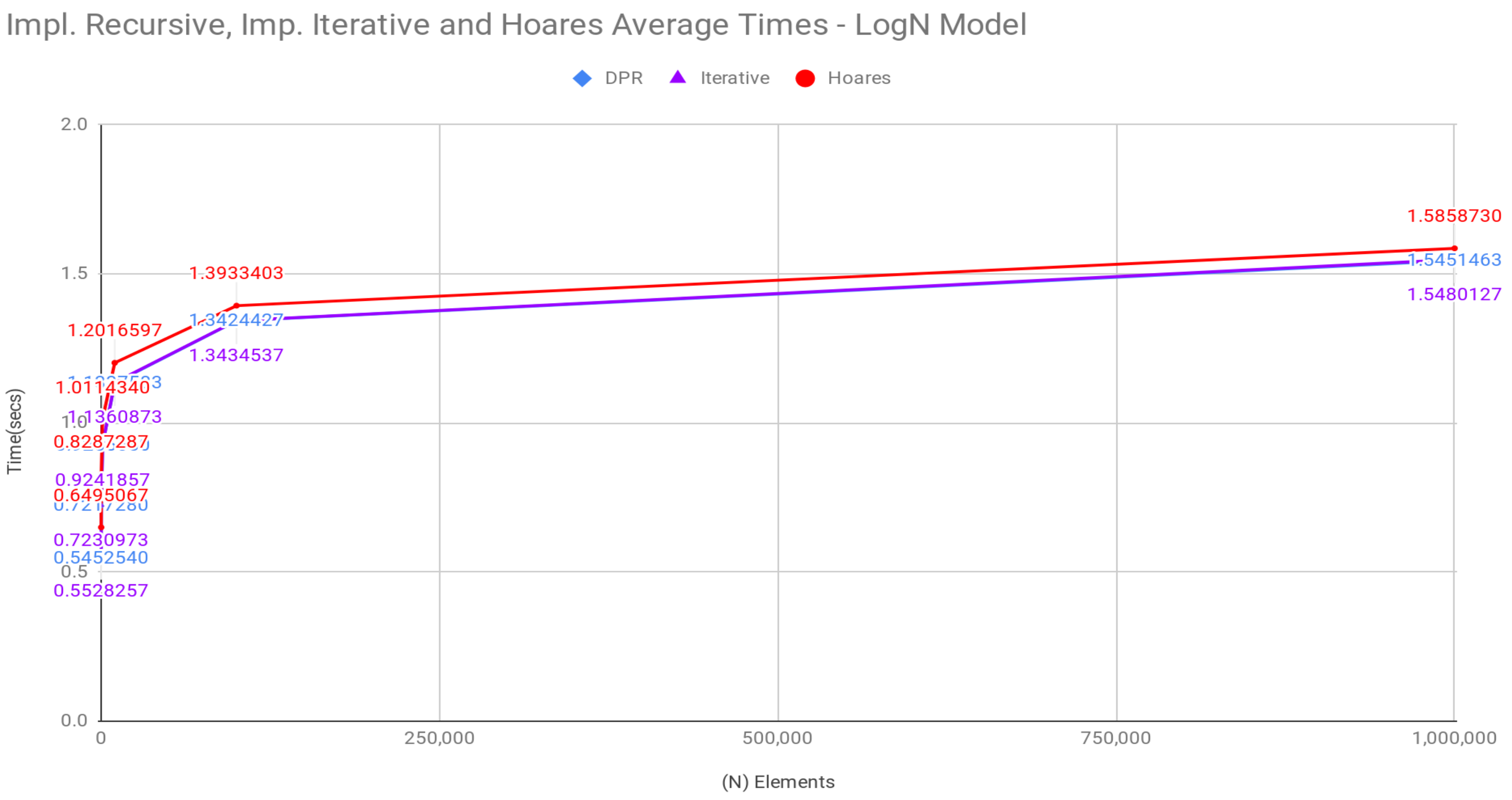
A3: Iterative with n array elements



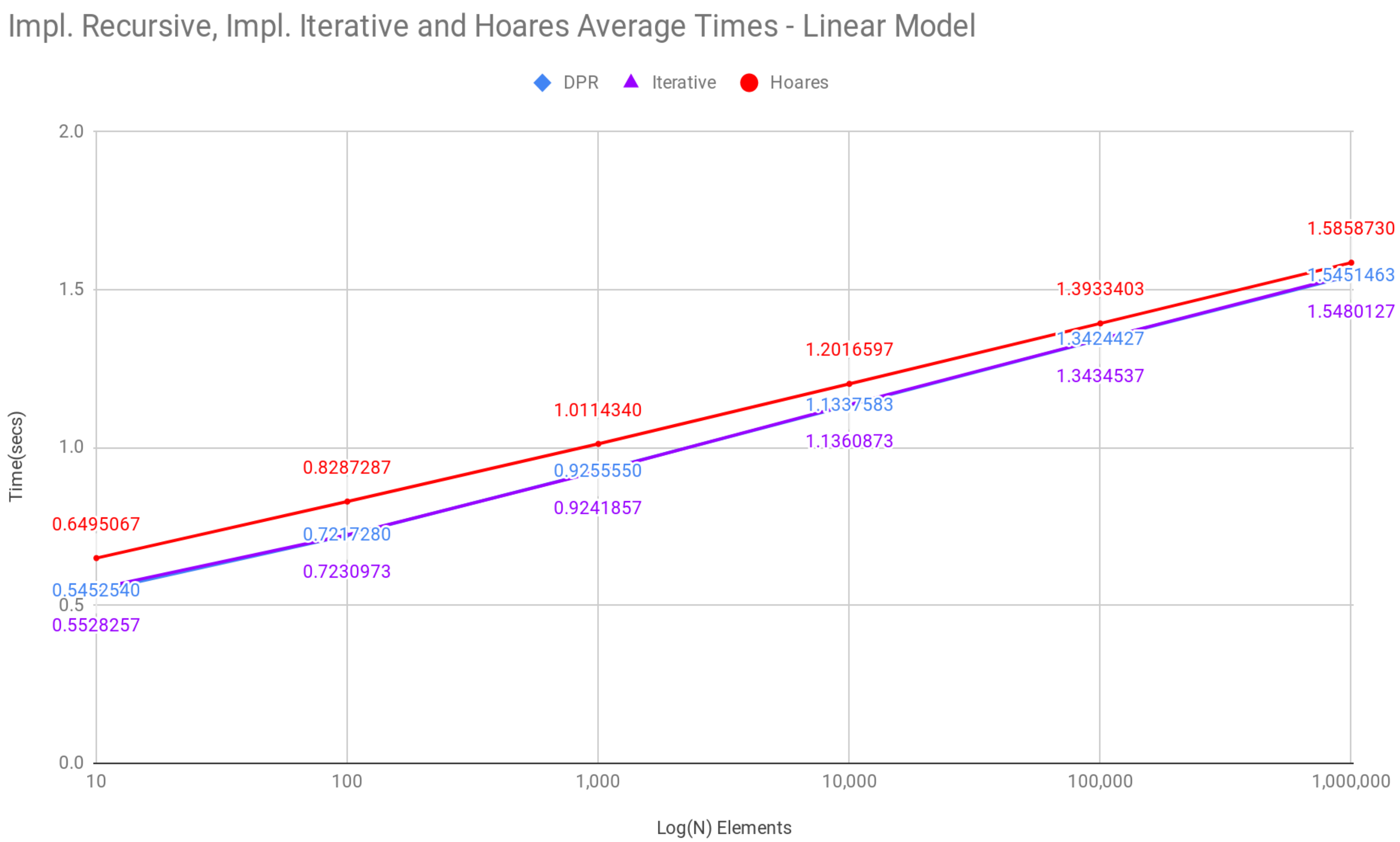
A3.1: Iterative with log n array elements



A4: Average of all n array elements



A4.1: Average of all with log n array elements



A5: Tabulated Data - Time in seconds

Impl. Rec	**run 1**	**run 2**	**run 3**		**AVG**
10,000,000	1.560824	1.539878	1.534737		1.5451463
1,000,000	1.339976	1.33754	1.349812		1.3424427
100,000	1.128875	1.130468	1.141932		1.1337583
10,000	0.925017	0.924568	0.92708		0.9255550
1,000	0.722005	0.721204	0.721975		0.7217280
100	0.54406	0.543991	0.547711		0.5452540
10	0.658074	0.656879	0.658755		0.6579027
Iterative	**run 1**	**run 2**	**run 3**		**AVG**
10,000,000	1.546849	1.545203	1.551986		1.5480127
1,000,000	1.340878	1.339286	1.350197		1.3434537
100,000	1.130027	1.131808	1.146427		1.1360873
10,000	0.923368	0.92367	0.925519		0.9241857
1,000	0.724961	0.721678	0.722653		0.7230973
100	0.550835	0.551613	0.556029		0.5528257
10	0.722427	0.719794	0.722414		0.7215450
Hoare's	**run 1**	**run 2**	**run 3**		**AVG**
10,000,000	1.57274	1.57997	1.604909		1.5858730
1,000,000	1.389575	1.386882	1.403564		1.3933403
100,000	1.195469	1.19674	1.21277		1.2016597
10,000	1.01061	1.010443	1.013249		1.0114340
1,000	0.828251	0.828488	0.829447		0.8287287
100	0.648045	0.648726	0.651749		0.6495067
10	0.685626	0.682919	0.685292		0.6846123

A6: Raw Data

See included .zip folder for raw data.