COMP2502 Assignment One

Algorithm Analysis

Ned Martin - 40529927
5 September 2004

ASYMPTOTIC ANALYSIS

SISorter Sort

The algorithm SISort, as implemented here, is the same as "Straight Insertion Sort[1]", the simplest form of insertion sorting, an algorithm that takes an initial sequence, and sorts it by placing the element in the i^{th} position into a second sequence, in sorted order. A linear search is used to locate the position at which the next element is to be inserted[1].

"Altogether, $n-1$ non-trivial insertions are required to sort a list of n elements.[2]"

"The number of iterations of the inner loop in the i^{th} iteration of the outer loop depends on the […] array [being sorted]. In the best case, the value in the position i of the array is larger than the value in position $i-1$ and zero iterations of the inner loop are done.[3]" In other words, the **best-case running time** is when the array is already sorted, resulting in a running time of $O(n)$. Conversely, for an array that has its elements already sorted, but in reverse order, i iterations of the inner loop are required in the i^{th} iteration of the outer loop[4], resulting in a **worst-case running time** of $O(n^2)$.

Now, consider all possible permutations of a sequence containing no duplicates. It turns out that the *average number of inversions in a permutation of n distinct elements is n(n-1)/4*[5]. If we consider that the act of sorting a list is actually the same as removing inversions, and that the inner loop of an insertion sort is actually removing insertions one at a time and that a swap takes a constant amount of time, then we see that the **average running time** for the SISort algorithm is $O(n^2)$[5].

StandardFirstSorter Sort

The algorithm StandardFirstSorter is a form of exchange sorting algorithm known as Quicksort[6]. Quicksort takes an unsorted sequence, selects a point known as the pivot point, and splits the sequence at this point into two sub-sequences. All elements less than or equal to the pivot point are placed into one sub-sequence, and all elements greater than or equal to the pivot are placed into the other sub-sequence. Quicksort is then recursively called on each of the sub-sequences, eventually resulting in a sorted sequence.

The **worst-case running time**, $O(n^2)$, occurs when one of the sub-sequences is empty, and the other contains all the remaining elements. Conversely, the **best-case running time**, $O(n \log_2(n))$, occurs when the sub-sequences are evenly populated. Using a little analytical logic, we can see that on average, if we randomly choose the pivot, we will end up with evenly-balanced sub-sequences as in the best-case running time, thus the **average running time** is also $O(n \log_2(n))$.

As it turns out, StandardFirstSorter always chooses the leftmost (in this case, first) element in a sequence as the pivot, giving it a **best-case running time** equivalent to its **worst-case running time** if the sequence is already ordered[7].

MedianOfThreeSorter Sort

The MedianOfThreeSorter algorithm is another implementation of the Quicksort algorithm explained in StandardFirstSorter above. In fact, the only difference is that this algorithm uses the *median-of-three pivot selection technique*[7] to select a pivot that is highly likely to be random, even for a sorted sequence, rather than choosing the leftmost element. As this pivot choosing technique runs in constant time, this gives the algorithm an **average running time** equivalent to its **best-case running time**, $O(n \log_2(n))$. It's worst-case running time remains the same, $O(n^2)$.

[1] B.R. Priess, *Data Structures and Algorithms with Object-Oriented Design Patterns in* Java, Fairfield: John Wiley & Sons, 2000, pp. 495
[2] B.R. Priess, Data Structures and Algorithms with Object-Oriented Design Patterns in Java, Fairfield: John Wiley & Sons, 2000, pp. 494
[3] B.R. Priess, *Data Structures and Algorithms with Object-Oriented Design Patterns in* Java, Fairfield: John Wiley & Sons, 2000, pp. 496
[4] B.R. Priess, *Data Structures and Algorithms with Object-Oriented Design Patterns in* Java, Fairfield: John Wiley & Sons, 2000, pp. 496
[5] B.R. Priess, Data Structures and Algorithms with Object-Oriented Design Patterns in Java, Fairfield: John Wiley & Sons, 2000, pp. 497
[6] B.R. Priess, *Data Structures and Algorithms with Object-Oriented Design Patterns in* Java, Fairfield: John Wiley & Sons, 2000, pp. 501
[7] B.R. Priess, *Data Structures and Algorithms with Object-Oriented Design Patterns in* Java, Fairfield: John Wiley & Sons, 2000, pp. 509

ALGORITHM TESTING

To test the above asymptotic analysis, an automated testing program[1] was written using Java. This program creates a sequence of integers of a given length, and runs them through a given sorting algorithm, outputting the time taken to sort the integers. The program accepts various arguments, providing a simple way to test a given sorting algorithm with a given set of data a given amount of times, averaging the results to prevent spurious differences in processing from skewing the timing. The results of these tests are presented and analysed in the next section.

Command line arguments for the program are as follows:

```
-(q|p) AVERAGE (1|2|3) (1|2|3) N {(1|2|3) (1|2|3) N}
-n AVERAGE (1|2|3) (1|2|3) N {N}
-m AVERAGE (1|2|3) (1|2|3) N {N} MULTIPLIER
```

Where AVERAGE is the number of times to run the sort, over which the result is averaged, N is the length of the sequence to sort and MULTIPLIER is a value by which each N is multiplied. The first (1|2|3) identifies the sequence used, 1 for random, two for ordered, 3 for reverse, and the second (1|2|3) specifies the sorting algorithm: 1 for StandardFirstSorter, 2 for MedianOfThreeSorter and 3 for SISorter. Specifying the -p flag will print the sequence used, both before and after sorting. Output from the program is formatted as comma separated values, suitable for manipulation by programs such as Microsoft Excel[2].

In producing the results below each algorithm was run a minimum of ten times, and the average time taken to sort the data recorded. An ordered sequence is specified as the set {1..N}, while a reverse sequence is {N..1}. A random sequence is a pseudo-random sequence of length N, with care taken to ensure that the sequence was the same for all tests within a given test case. A random sequence may contain duplicates but is unlikely to.

[1] Java Program Attached to this submission.
[2] Microsoft Excel, Microsoft Corporation,

Test Results

The results of testing the sorting algorithms StandardFirstSorter, MedianOfThreeSorter and SISorter were recorded for a series of values of N such that there was significant difference between the highest and lowest timings. In general, an exponentially increasing N was used, beginning at N high enough to produce a statistically valid output – generally found to be around 10 to 20 milliseconds as values lower than this were too subject to small differences in the computer's processing, and increasing until a either a statistically significant range was generated or the limitations of the JVM were reached. Tables of values were then created in Excel, graphed[1], and various values calculated, as shown below. Note that only seven median values have been shown for each algorithm below. Many more were actually calculated for the graphing.

[1] Graphs available in Appendix A.

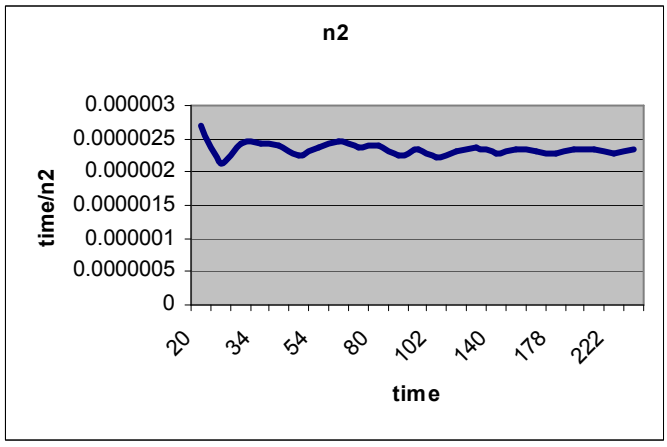
Test Analysis

StandardFirstSorter

Reverse StandardSort

n	$\log_2(n)$	$n \log_2(n)$	n^2	time	time/n	time/$\log_2(n)$	time/$n \log_2(n)$	time/n^2
1360	10.4094	14156.8	1849600	8	0.00588	0.76854	0.00057	4.3E-06
2720	11.4094	31033.5	7398400	20	0.00735	1.75294	0.00064	2.7E-06
4080	11.9944	48937	1.7E+07	40	0.0098	3.3349	0.00082	2.4E-06
5440	12.4094	67507.1	3E+07	70	0.01287	5.64089	0.00104	2.4E-06
6800	12.7313	86573	4.6E+07	102	0.015	8.01174	0.00118	2.2E-06
8160	12.9944	106034	6.7E+07	156	0.01912	12.0052	0.00147	2.3E-06
9520	13.2167	125823	9.1E+07	212	0.02227	16.0403	0.00168	2.3E-06

Based on the above data, it is clear that the ratio $time/n^2$ is approaching a constant, indicating that this algorithm's average running time for a sequence of reversed integers is $O(n^2)$. This can perhaps be seen more clearly in the graph[1] of this function, below.

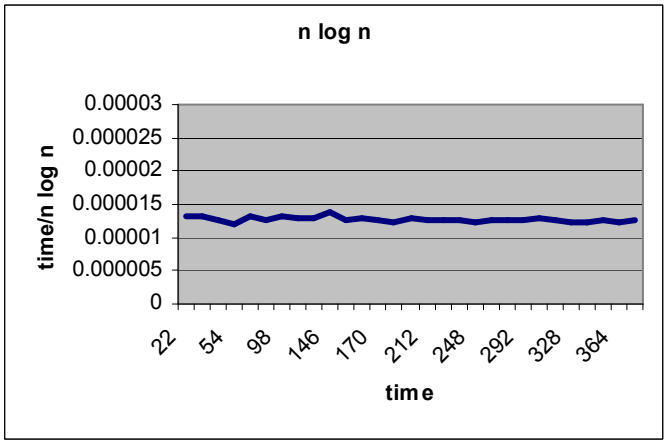


Here we see the ratio $time/n^2$ graphed against $time$, where $time$ is the time taken for the algorithm to sort the given data, and n is the amount of data being sorted. A flat graph shows that this ratio is tending towards a constant, which, as shown by [2], is the constant c, in the formula for "big oh". Graphs clearly show whether the ratios converge, diverge, or tend towards a constant.

Random StandardSort

n	$\log_2(n)$	$n \log_2(n)$	n^2	time	time/n	time/$\log_2(n)$	time/$n \log_2(n)$	time/n^2
150000	17.1946	2579190	2.3E+10	34	0.00023	1.97736	1.3E-05	1.5E-09
350000	18.417	6445948	1.2E+11	82	0.00023	4.45241	1.3E-05	6.7E-10
550000	19.0691	1E+07	3E+11	146	0.00027	7.65638	1.4E-05	4.8E-10
750000	19.5165	1.5E+07	5.6E+11	182	0.00024	9.32543	1.2E-05	3.2E-10
950000	19.8576	1.9E+07	9E+11	240	0.00025	12.0861	1.3E-05	2.7E-10
1150000	20.1332	2.3E+07	1.3E+12	292	0.00025	14.5034	1.3E-05	2.2E-10
1350000	20.3645	2.7E+07	1.8E+12	338	0.00025	16.5975	1.2E-05	1.9E-10

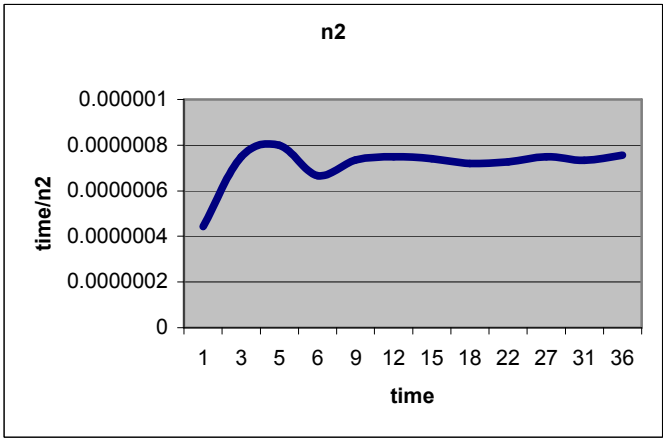
Once again, one ratio clearly approaches a constant, indicating that StandardFirstSorter sorts sequences of random integers in $O(n \log_2(n))$. This can be seen more clearly in the graph of this function.



Ordered StandardSort

n	$\log_2(n)$	$n \log_2(n)$	n^2	time	time/n	time/$\log_2(n)$	time/$n \log_2(n)$	time/n^2
1500	10.55075	15826.12	2250000	1	0.000667	0.09478	6.32E-05	4.44E-07
2500	11.28771	28219.28	6250000	5	0.002	0.44296	0.000177	8E-07
3500	11.77314	41205.99	12250000	9	0.002571	0.764452	0.000218	7.35E-07
4500	12.13571	54610.69	20250000	15	0.003333	1.236022	0.000275	7.41E-07
5500	12.42522	68338.69	30250000	22	0.004	1.770593	0.000322	7.27E-07
6500	12.66622	82330.46	42250000	31	0.004769	2.447454	0.000377	7.34E-07
6899	12.75217	87977.23	47596201	36	0.005218	2.823049	0.000409	7.56E-07

StandardFirstSorter, based on the above data, appears to sort already ordered data in $O(n^2)$.



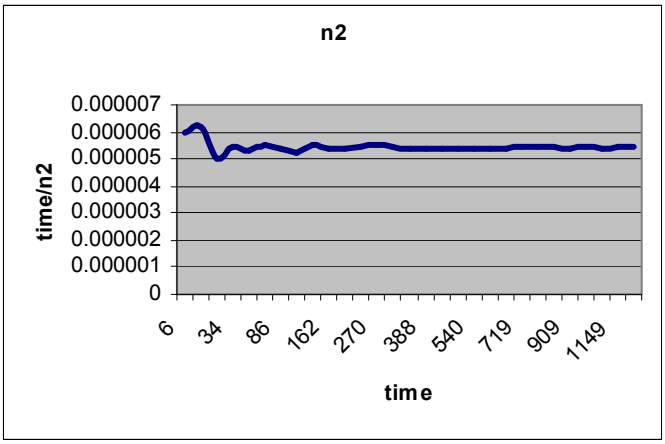
For most of the test cases, values of n were chosen that produced values for *time* ranging from 10 milliseconds upwards, as values below 10 were found to be too easily influenced by small unpreventable changes in the operating environment of the computer running the test cases. This can be clearly seen in the values of *time* below 10 on this graph. The StandardFirstSorter, due to its recursive nature, causes a stack overflow error when sorting large amounts of data, thus preventing an n high enough to achieve *time* values above 36 milliseconds. To partially overcome this limitation, this particular test was run several times, averaged over one hundred times, and the results of each of these tests then averaged to offset changes caused by the uncontrollable variances in the machine running the tests.

SISorter

Reverse SISorter

n	$\log_2(n)$	$n \log_2(n)$	n^2	time	time/n	time/$\log_2(n)$	time/$n \log_2(n)$	time/n^2
1500	10.5507	15826.1	2250000	14	0.00933	1.32692	0.00088	6.2E-06
3500	11.7731	41206	1.2E+07	68	0.01943	5.77586	0.00165	5.6E-06
5500	12.4252	68338.7	3E+07	162	0.02945	13.038	0.00237	5.4E-06
7500	12.8727	96545.1	5.6E+07	308	0.04107	23.9267	0.00319	5.5E-06
9500	13.2137	125530	9E+07	488	0.05137	36.9313	0.00389	5.4E-06
11500	13.4893	155127	1.3E+08	719	0.06252	53.3013	0.00463	5.4E-06
13500	13.7207	185229	1.8E+08	989	0.07326	72.081	0.00534	5.4E-06

The ratio of $time/n^2$ clearly approaches a constant for reverse-ordered data sorted by SISorter, suggesting that this algorithm is running in $O(n^2)$ time, and this can also be seen clearly in the graph below.

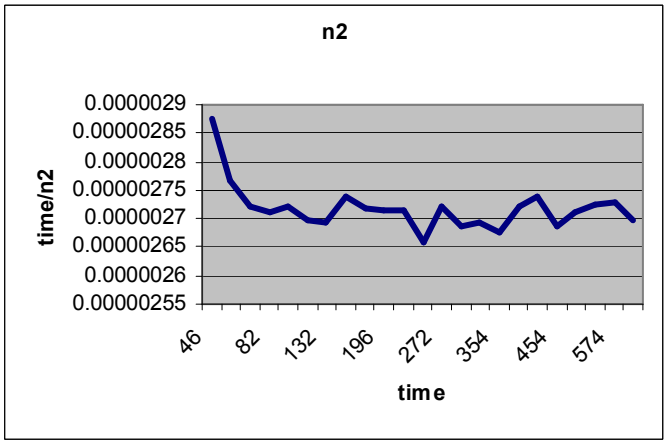


If we remember, SISorter will run its worst-case running time of $O(n^2)$ unless the data is already sorted. Considering that reverse-ordered data is about as far from sorted as possible, this result is to be expected.

Random SISorter

n	$\log_2(n)$	$n \log_2(n)$	n^2	time	time/n	time/$\log_2(n)$	time/$n \log_2(n)$	time/n^2
1500	10.55075	15826.12	2250000	8	0.005333	0.75824	0.000505	3.56E-06
3500	11.77314	41205.99	12250000	30	0.008571	2.548173	0.000728	2.45E-06
5500	12.42522	68338.69	30250000	82	0.014909	6.599483	0.0012	2.71E-06
7500	12.87267	96545.06	56250000	154	0.020533	11.96333	0.001595	2.74E-06
9500	13.21371	125530.3	90250000	240	0.025263	18.16295	0.001912	2.66E-06
11500	13.48935	155127.5	1.32E+08	354	0.030783	26.24293	0.002282	2.68E-06
13500	13.72067	185229.1	1.82E+08	494	0.036593	36.00407	0.002667	2.71E-06

Sorting randomly ordered data with SISorter gives us a running time of n^2, as shown by the $time/n^2$ column being roughly constant, and its graph approaching a constant.

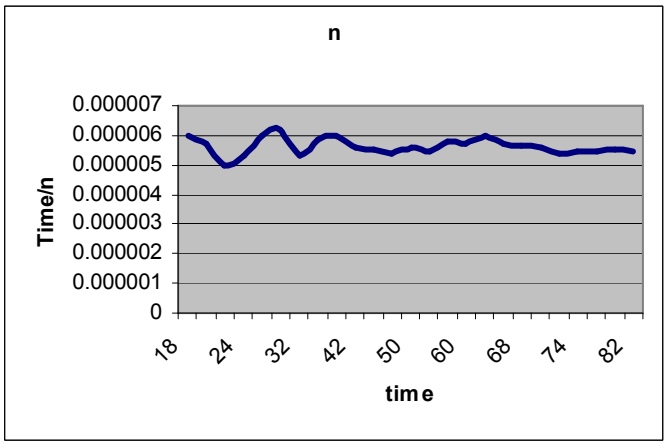


Once again, this is what we expect.

Ordered SISorter

n	$\log_2(n)$	$n \log_2(n)$	n^2	time	time/n	time/$\log_2(n)$	time/$n \log_2(n)$	time/n^2
1500000	20.51653	30774797	2.25E+12	8	5.33E-06	0.389929	2.6E-07	3.56E-12
3500000	21.73892	76086232	1.23E+13	20	5.71E-06	0.920009	2.63E-07	1.63E-12
5500000	22.391	1.23E+08	3.03E+13	34	6.18E-06	1.518467	2.76E-07	1.12E-12
7500000	22.83846	1.71E+08	5.63E+13	42	5.6E-06	1.839003	2.45E-07	7.47E-13
9500000	23.1795	2.2E+08	9.03E+13	52	5.47E-06	2.243362	2.36E-07	5.76E-13
11500000	23.45513	2.7E+08	1.32E+14	66	5.74E-06	2.813883	2.45E-07	4.99E-13
13500000	23.68646	3.2E+08	1.82E+14	74	5.48E-06	3.124148	2.31E-07	4.06E-13

Sorting ordered data using SISorter gives us two ratios that approach a constant, $time/(n \log_2(n))$ and $time/n$, giving us two possible "big oh" values.

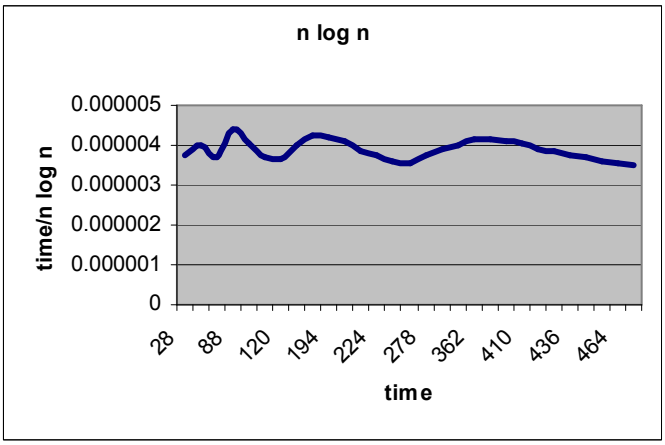


Further analysis, however, will reveal that sorting already sorted data is the best-case for SISorter, which happens to be $O(n)$, so we conclude that $O(n)$ is indeed the correct value.

MedianOfThreeSorter

Reverse MedianOfThree								
n	$\log_2(n)$	$n \log_2(n)$	n^2	time	time/n	time/$\log_2(n)$	time/$n \log_2(n)$	time/n^2
800000	19.61	2E+07	6E+11	58	7E-05	2.9577	3.76E-06	9.063E-11
2E+06	20.61	3E+07	3E+12	120	8E-05	5.8225	4.42E-06	4.688E-11
2E+06	21.195	5E+07	6E+12	208	9E-05	9.8138	3.71E-06	3.611E-11
3E+06	21.61	7E+07	1E+13	244	8E-05	11.291	3.64E-06	2.383E-11
4E+06	21.932	9E+07	2E+13	362	9E-05	16.506	4.13E-06	2.263E-11
5E+06	22.195	1E+08	2E+13	418	9E-05	18.833	3.92E-06	1.814E-11
6E+06	22.417	1E+08	3E+13	454	8E-05	20.252	3.51E-06	1.448E-11

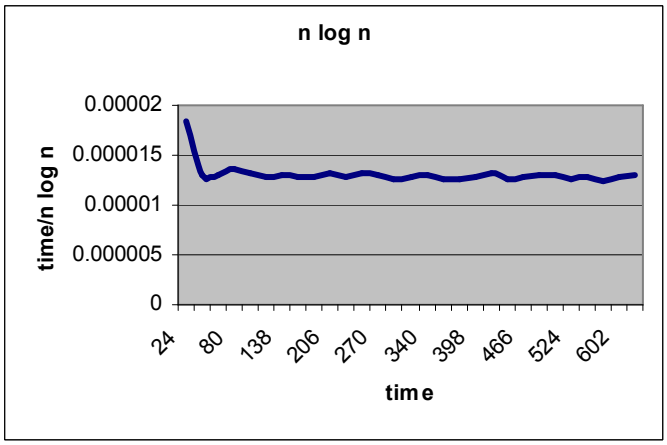
Examining the different ratios of time over various possible "big oh" values, we can conclude that sorting reverse-ordered data with the MedianOfThreeSorter algorithm, is either $O(n \log_2(n))$ or $O(n)$.



Given that the best-case running time for the MedianOfThreeSorter, as given in the Asymptotic Analysis at the start, is $O(n \log_2(n))$, we can assume that it will not run better than its best – thus $O(n \log_2(n))$ is the plausible value here.

Random MedianOfThree								
n	$\log_2(n)$	$n \log_2(n)$	n^2	time	time/n	time/$\log_2(n)$	time/$n \log_2(n)$	time/n^2
320000	18.2877	5852068	1E+11	80	0.00025	4.37452	1.37E-05	7.8E-10
640000	19.2877	1.2E+07	4.1E+11	158	0.00025	8.19174	1.31E-05	3.9E-10
960000	19.8727	1.9E+07	9.2E+11	252	0.00026	12.6807	1.32E-05	2.7E-10
1280000	20.2877	2.6E+07	1.6E+12	340	0.00027	16.7589	1.29E-05	2.1E-10
1600000	20.6096	3.3E+07	2.6E+12	432	0.00027	20.9611	1.28E-05	1.7E-10
1920000	20.8727	4E+07	3.7E+12	518	0.00027	24.8171	1.3E-05	1.4E-10
2240000	21.0951	4.7E+07	5E+12	602	0.00027	28.5375	1.26E-05	1.2E-10

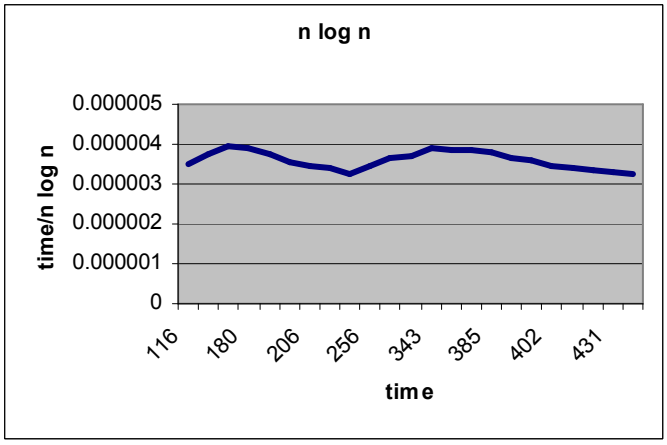
Once again we see the unreliability of small *time* values towards the lower end of this graph.



Here we can come to the same conclusion as above: the average running-time using randomly ordered data with this algorithm is $O(n \log_2(n))$.

Ordered MedianOfThree								
n	$\log_2(n)$	$n \log_2(n)$	n^2	time	time/n	time/$\log_2(n)$	time/$n \log_2(n)$	time/n^2
1000000	19.9316	2E+07	1E+12	78	7.8E-05	3.91339	3.9E-06	7.8E-11
1800000	20.7796	3.7E+07	3.2E+12	140	7.8E-05	6.73739	3.7E-06	4.3E-11
2600000	21.3101	5.5E+07	6.8E+12	196	7.5E-05	9.19753	3.5E-06	2.9E-11
3400000	21.6971	7.4E+07	1.2E+13	256	7.5E-05	11.7988	3.5E-06	2.2E-11
4200000	22.002	9.2E+07	1.8E+13	354	8.4E-05	16.0895	3.8E-06	2E-11
5000000	22.2535	1.1E+08	2.5E+13	401	8E-05	18.0196	3.6E-06	1.6E-11
5800000	22.4676	1.3E+08	3.4E+13	431	7.4E-05	19.1832	3.3E-06	1.3E-11

Similarly to above, the MedianOfThreeSorter algorithm sorts ordered data in $O(n \log_2(n))$ time, this being it's best-case, and average, running time.



[1] See graph in Appendix A
[2] B.R. Priess, *Data Structures and Algorithms with Object-Oriented Design Patterns in* Java, Fairfield: John Wiley & Sons, 2000, pp. 36

CONCLUSION

In conclusion, we have found that the original asymptotic analysis is well-supported by the test evidence, with all tests producing the running time averages we would expect.

The **SISorter algorithm** ran at its theoretical best, $O(n)$, for ordered data, and at its average (and also theoretical worst) $O(n^2)$ time for both reverse-ordered and randomly-ordered data. This is to be expected, given the way that algorithm works.

The **StandardFirstSorter algorithm** ran at its worst, $O(n^2)$, for both ordered and reverse-ordered data, but ran at its average $O(n \log_2(n))$ when sorting randomly-ordered data. If we remember, SISorter is a Quicksort algorithm that chooses the leftmost element as its pivot, thus losing efficiency on ordered data, but when the data is randomly-ordered choosing any pivot is equally efficient.

The **MedianOfThreeSorter algorithm** ran at its average $O(n \log_2(n))$ time for all test data, ordered or not. This makes it the most efficient overall sorting algorithm of the three tested.

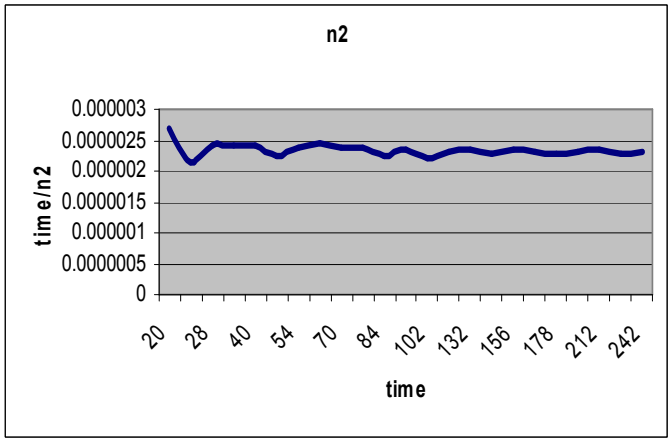
The SISorter algorithm provides the most efficient way to sort already sorted data – a very questionable advantage, but is inefficient on other data. Of the two Quicksort algorithms tested, the StandardFirstSorter algorithm is efficient only on randomly-ordered data, while the MedianOfThreeSorter algorithm is equally efficient when sorting ordered or non-ordered data, making it the clear choice for sorting any data that isn't already sorted.

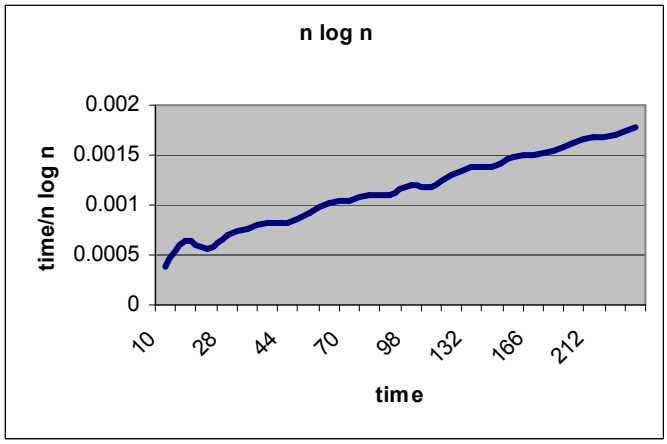
Notes

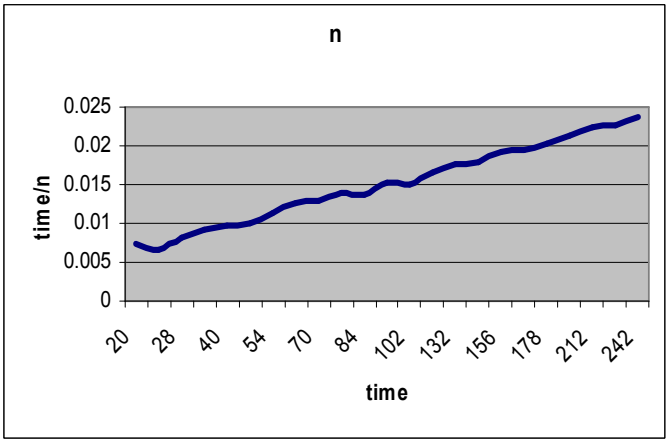
- Tests were conducted on a Pentium 4 2.8 GHz computer with 512 MB RAM running Windows XP.
- The Java version used was "1.3.1"
  Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1-b24)
  Java HotSpot(TM) Client VM (build 1.3.1-b24, mixed mode)
- Values in tables have been rounded to only a few significant digits for brevity, and only median values are shown. The actual values used in calculation and graphing were calculated to the maximum precision provided by Microsoft Excel.
- The randomly-ordered data used throughout was generated by manually seeding Java's Random() function, which will then generate a pseudorandom sequence using a linear congruential formula. Random sequences may contain duplicates, but are unlikely to for small sequences. In testing, the same random sequence was used for each test within a single averaged operation, but a different random sequence was used for any different test cases.

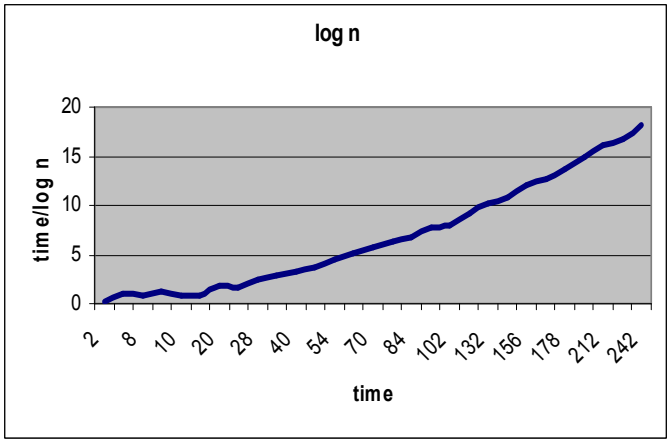
Appendix A – Algorithmic Analysis Graphs

Reverse Standard Sort

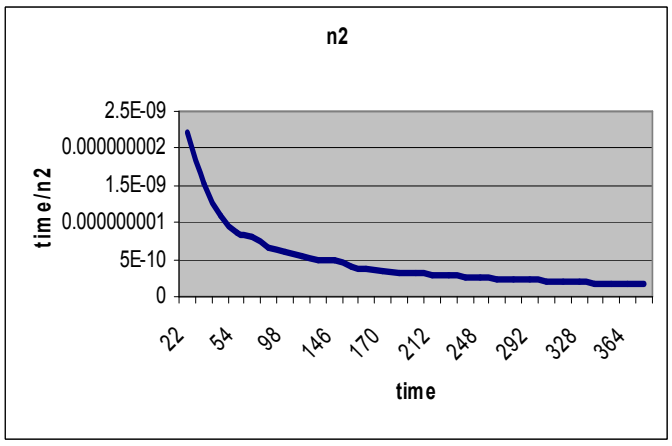


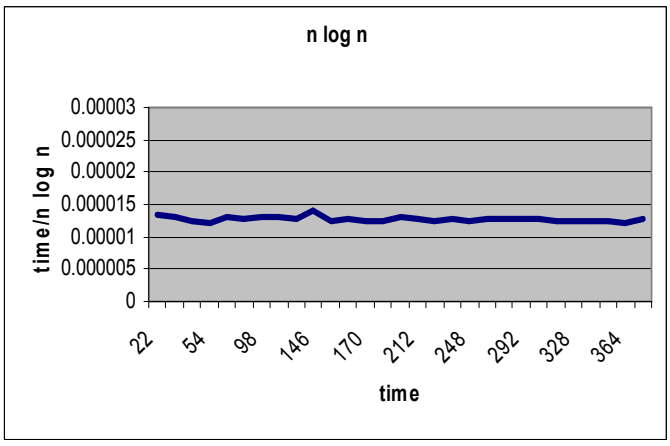


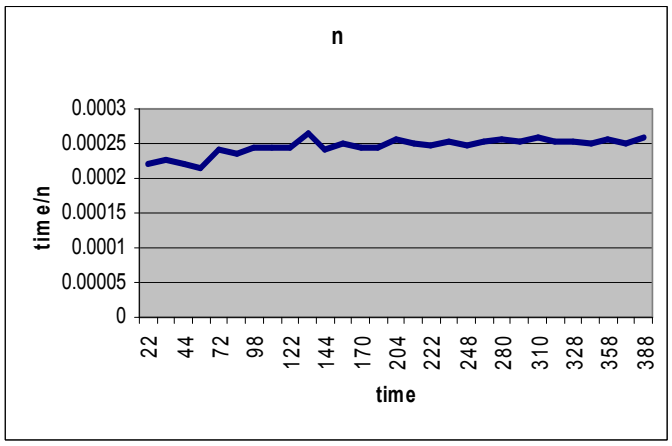


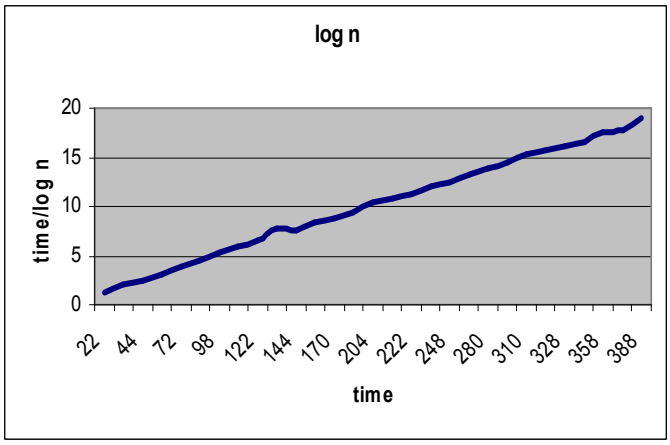


Random Standard Sort

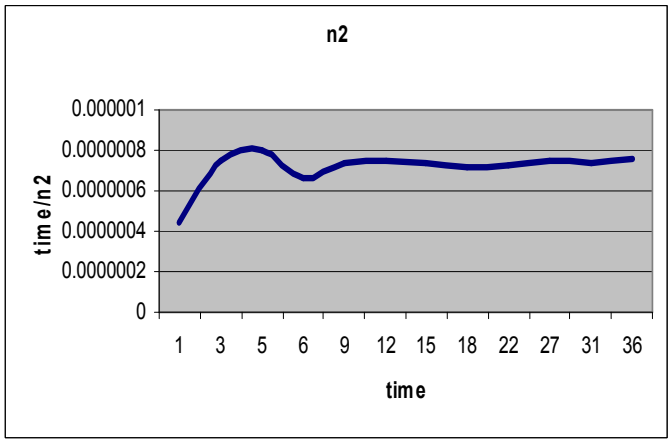


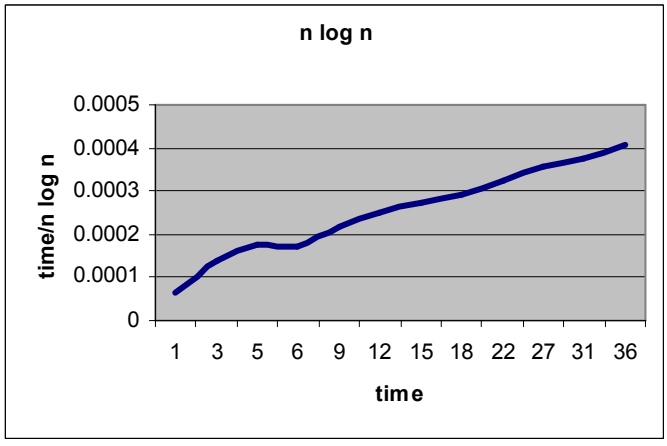


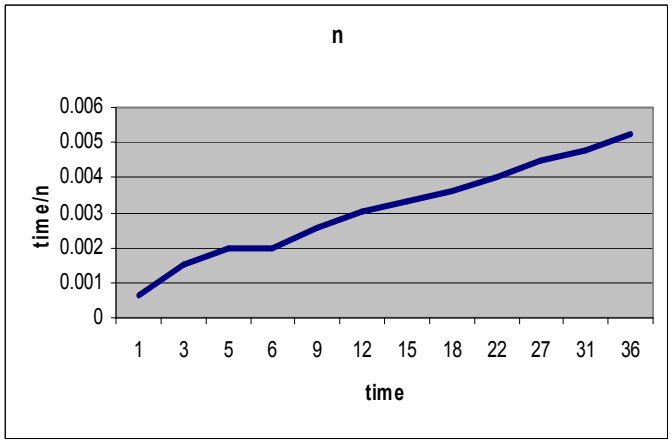


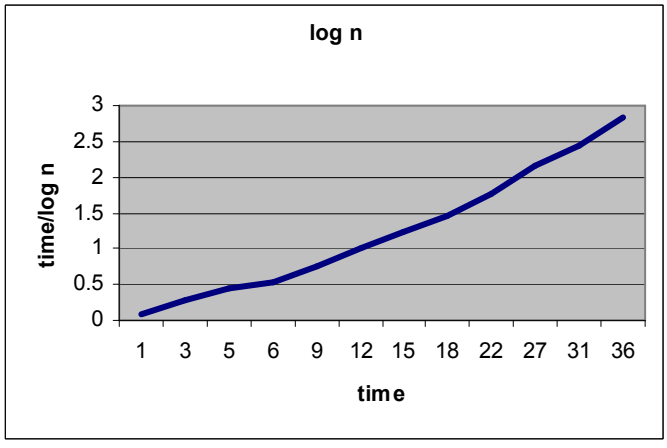


Ordered Standard Sort

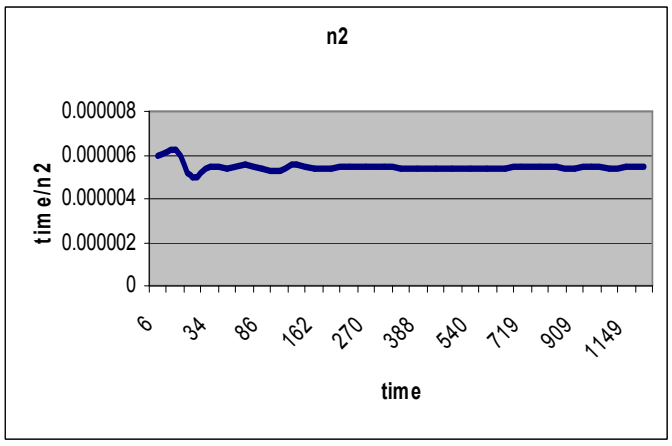


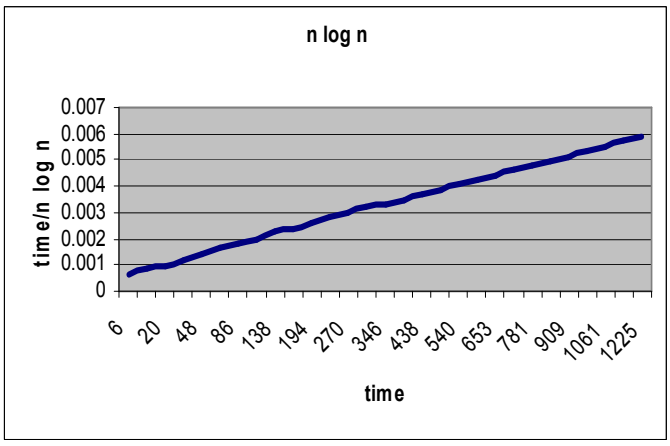


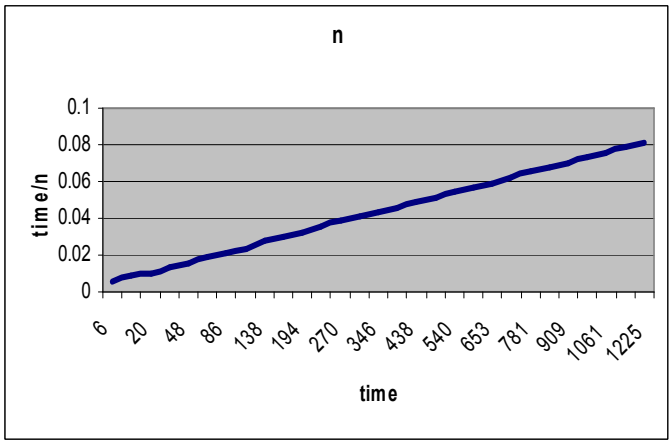


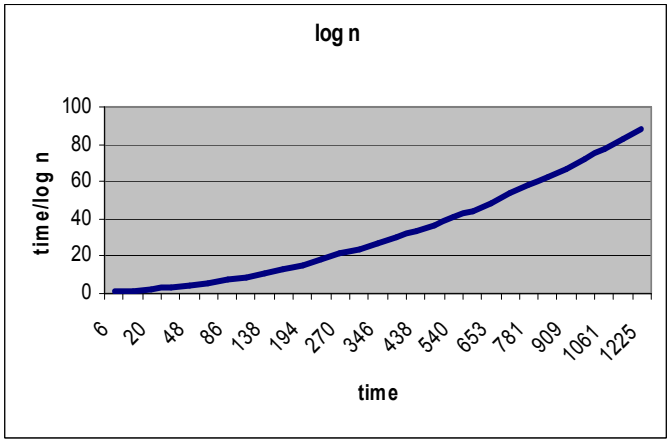


Reverse SISorter Sort

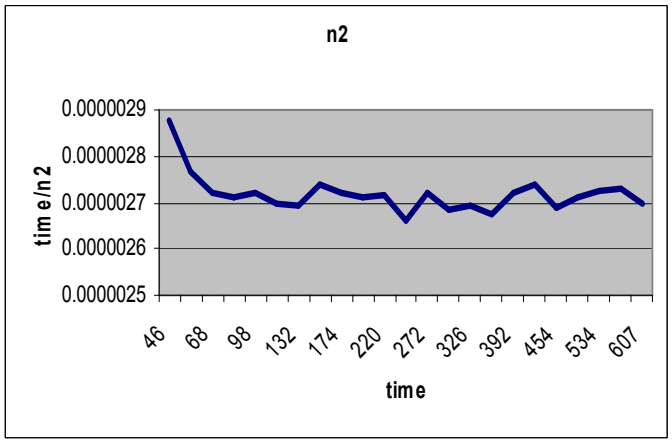


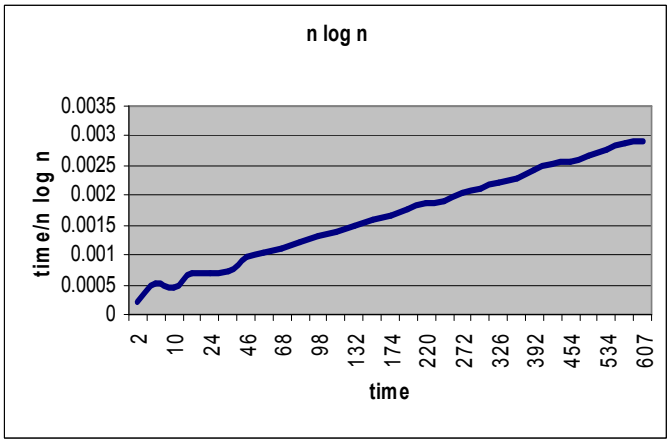


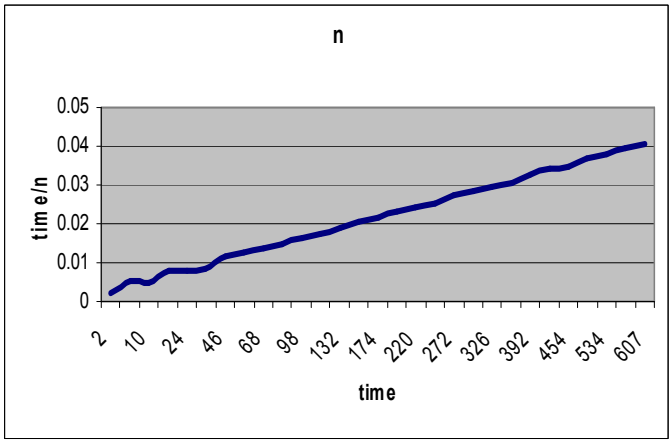


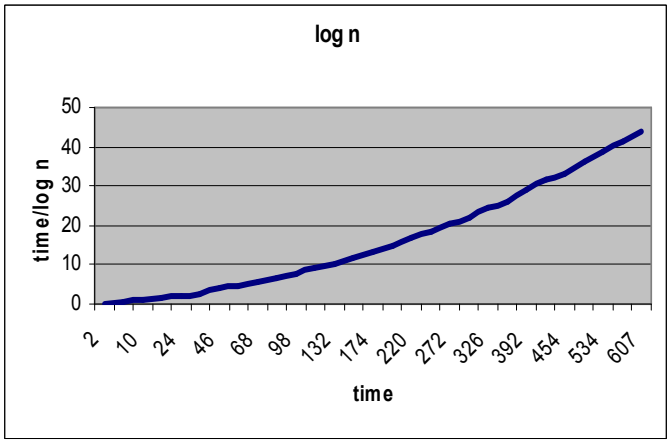


Random SISorter Sort

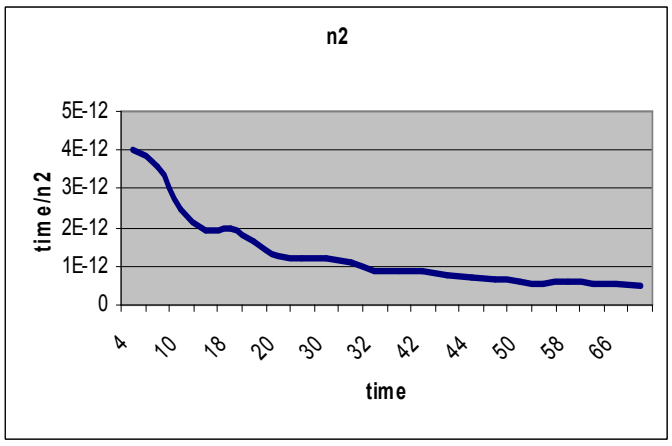


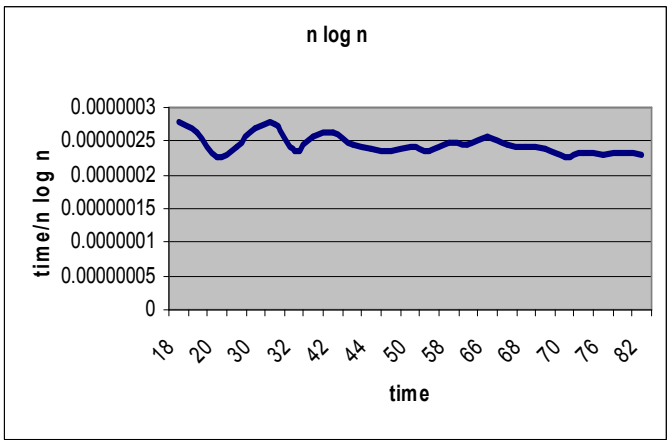


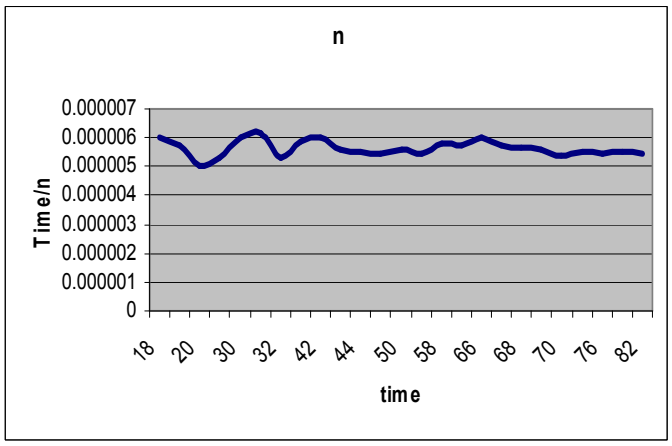


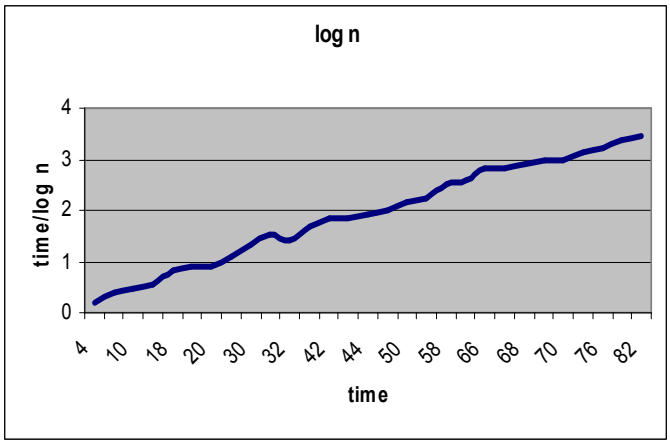


Ordered SISorter Sort

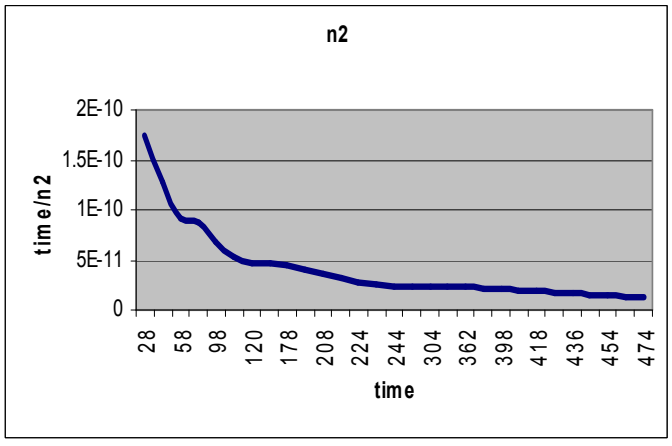


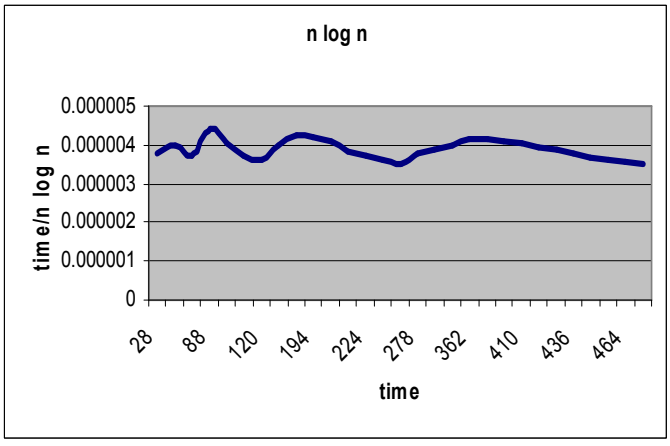


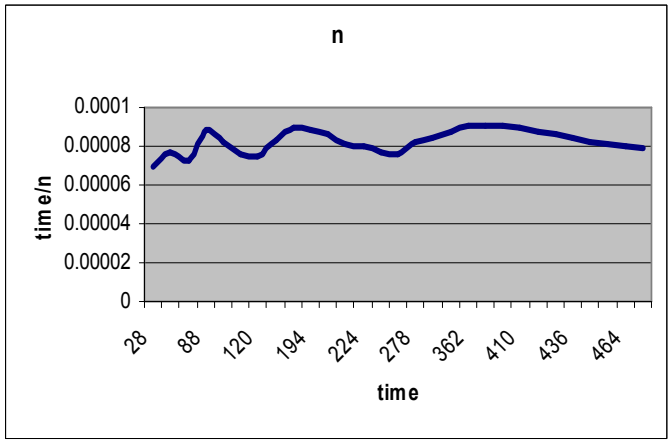


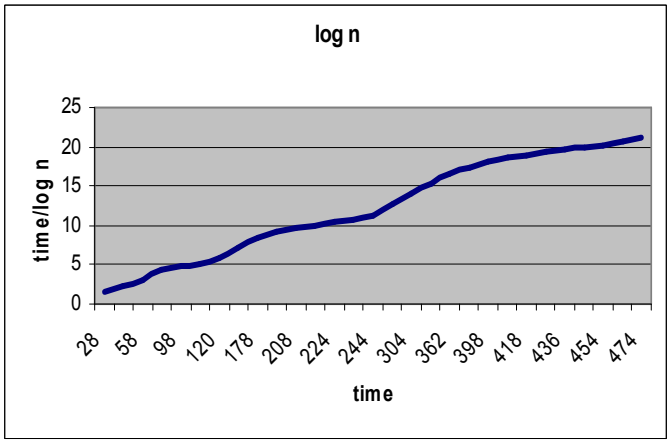


Reverse MedianOfThree Sort

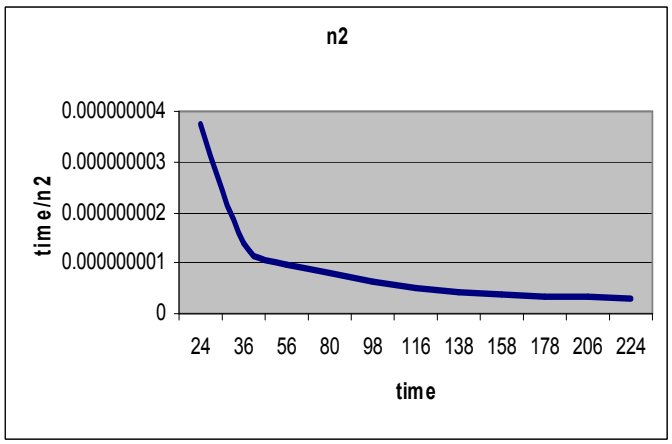


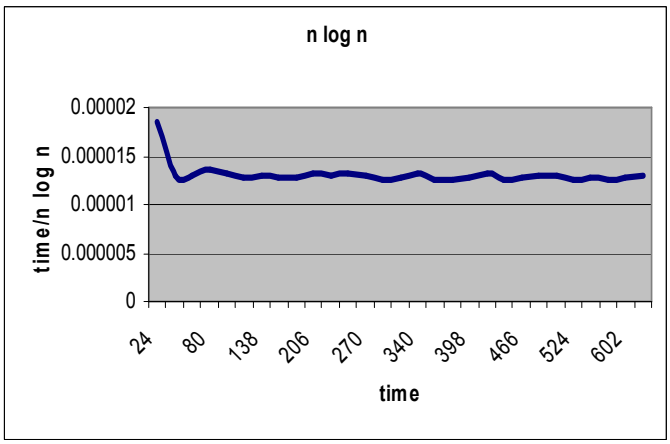


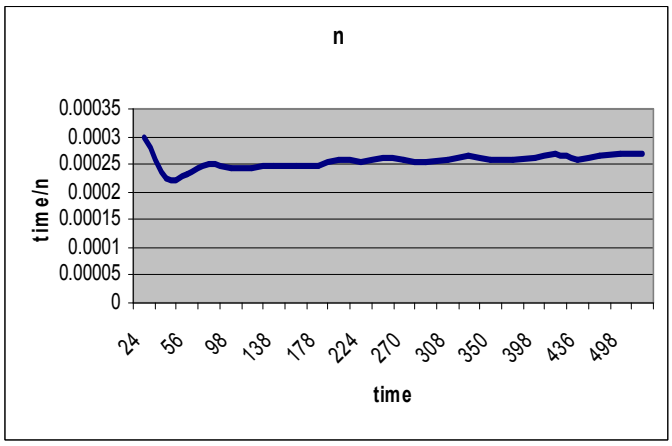


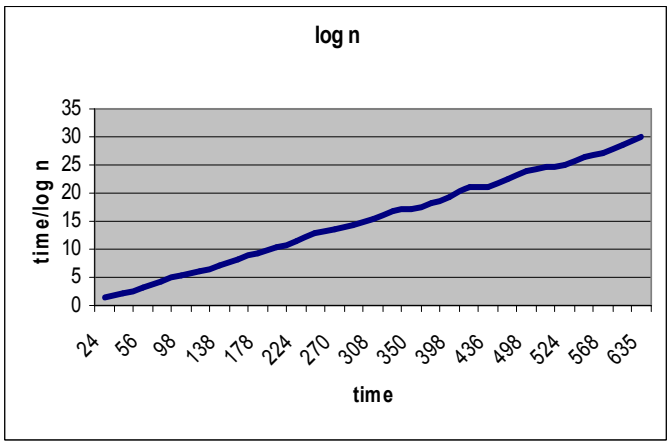
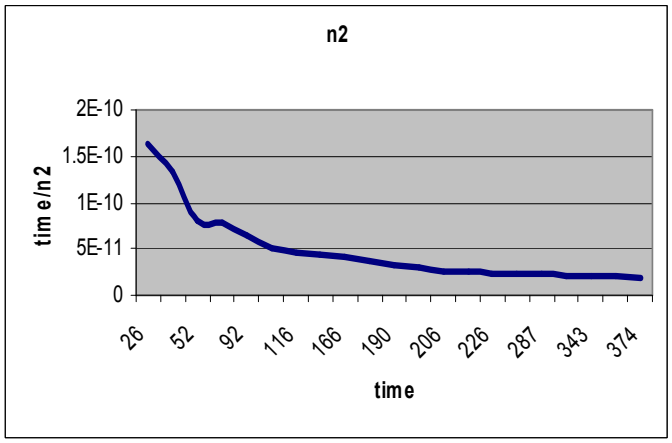
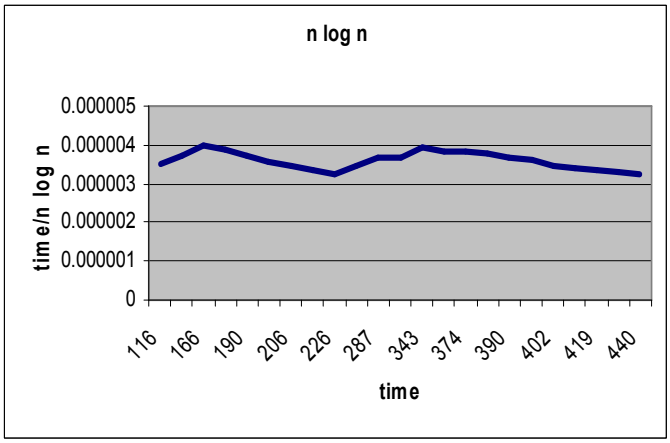
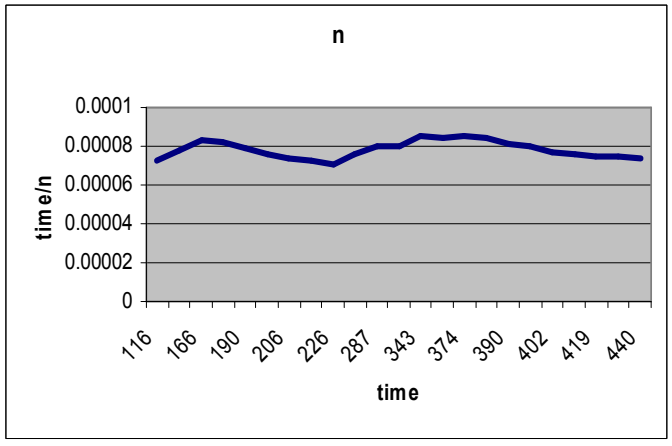
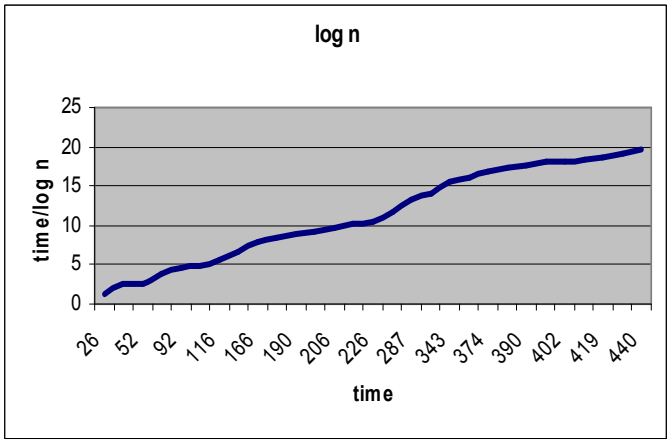


Random MedianOfThree Sort









Ordered MedianOfThree Sort