# Runtime of Two Implementations of the Quicksort Sorting Algorithm (Empirical Findings & Analysis)

**Steve Harman**

**Programming Assignment 1**

**CIS 680**

**2/19/2004**

## I. Introduction

The objective of this programming assignment was to show (empirically) that the average runtime of the Quicksort sorting algorithm is $\Theta(n \lg n)$, and that Randomized Quicksort will also run in $\Theta(n \lg n)$ time.

## II. Design & Implementation of Algorithms

The first thing to be done was designing and implementing the two Quicksort algorithms to be used. To accomplish this I simply implemented (using C++) the standard Quicksort and Partition algorithms discussed in class. Implementation of the Randomized Quicksort algorithm was a bit trickier. I began by trying to actually write a special "random partition" function which would randomly pick a pivot element, and then partition based on it. I was having some trouble with the actual implementation details and decided to try and find a better way of randomizing the partition action. After a little "Googling" I realized that it is much easier to simply pick a random element of the array, swap it with the last element of the array, and then proceed with the partition function used for standard Quicksort (1). To pick the random pivot element, I used the standard C++ *rand()* function, and then trimmed it down to a number between the current *p* and *r* of the array in question. I followed this with a simple *swap(array[random_index] , array[r])*, resulting in a random pivot element to be partitioned on.

The next task was to build the input arrays which were to be sorted. I decided that I would need two types of arrays for testing the runtimes of the algorithms: sorted and unsorted. Also, I would need to run both types of arrays through each of the two algorithms. This meant I needed a total of four arrays, 2 sorted and 2 unsorted, all of size *n* (which is taken from user input). The arrays were built by running through a loop *n* times, inserting an element at each loop. The elements of the sorted array went from *0* to *n-1*, and the elements of the unsorted arrays were just random integers generated by the *rand()* function. The sorted arrays were used to simulate "worst-case" inputs, while the unsorted (and ideally random) arrays simulated "average-case" inputs, for the Quicksort algorithms.

The last real implementation task was to make sure that both algorithms were run with both a sorted and an unsorted input array. A simple *while()* loop with a nested *switch ()* statement then built the needed arrays, set a Boolean variable (to determine whether or not to use Randomized Quicksort), and output the runtimes to the screen.

## III. Running Test Cases & Gathering Results

The next step was to run several sets of test cases with varying array sizes and record the runtimes for each test case. I started with a small array size (10) and worked up to much larger ones (65,000). The smaller cases ran very fast for both Quicksort algorithms, and for both sorted and unsorted arrays. However, as the array size started to grow an increase in the runtime became apparent, especially in the normal Quicksort algorithms. An example of the output can be seen below in Figure 1.1.

```
% lab1

How many items in the array? 40000
Normal Quicksort, Sorted:          size= 40000    time= 54.923
Random Quicksort, Sorted:          size= 40000    time= 0.0800559
Normal Quicksort, Unsorted:        size= 40000    time= 10.2817
Random Quicksort, Unsorted:        size= 40000    time= 0.0754752
Run Again (Y/N)?
```

Figure 1.1

In an attempt to gain an average runtime for each algorithm (with each of the given array sizes), I ran each algorithm with a particular array size three and compiled the results of each test case in an Excel spreadsheet. I then took an average of those three times (per algorithm, per array size) and used that as my "average runtime". Then I took each of the average runtimes into another Excel spreadsheet and graphed them to get a visual representation/comparison of the runtimes.

## *IV. Analyzing the Data & Conclusions*

In order to begin analyzing the data I needed to establish some reference data. To do this I also calculated the "time" to sort if the algorithms ran in $n \lg n$ and $n^2$ time. I also had to introduce a constant into these reference times to bring the numbers down so they were on par with my experimental results. I then graphed these new reference times alongside the experimental results for comparison. Upon looking at the data it seemed that Randomized Quicksort ran in approximately the same time for both sorted and unsorted input arrays. A quick look at the attached graph, "Quicksort Runtimes 2," will show that they seem to run at about $n \lg n$ time. (You will notice that I also created a second set of $n \lg n$ and $n^2$ data points where had to be reduced to get them down to the Randomized Quicksort times. These points appear on the "Quicksort Runtime 2" graph.) After some more inspection it became clear that Randomized Quicksort was out-performing the standard Quicksort algorithm by quite a lot. It appeared that starting with input arrays of just a few hundred elements, Randomized Quicksort really starts to shine. More precisely, it can be seen that given an already sorted input, the standard Quicksort algorithm preformed slower by a factor of $10^1$ with an array size of just 200 elements. With an unsorted input it took an array size of 500 to get this same performance drop. Given a sorted input of size 5,000, the standard Quicksort preformed worse by a factor of about $10^2$; this same result is achieved with an unsorted input size of about 5,000-10,000. With a sorted array of around 20,000 elements the standard Quicksort algorithm suffered a performance drop of a factor of nearly $10^3$. With unsorted input arrays of size 65,000, the performance drop was not quite a factor of $10^3$.

From the graph "Quicksort Runtimes" I was also able to conclude that the worst-case runtime for the standard Quicksort algorithm is indeed bound by $O(n^2)$. The graph also seems to show that given an average (that is, a unsorted and random) input, the standard Quicksort algorithm runs in about $n \ln n$, or more precisely $\Theta(n \lg n)$, time. This can be seen by looking at the "Quicksort Runtimes" graph and seeing how the orange (standard Quicksort with unsorted input) runs a very similar slope to the grey ($n \lg n$) line.

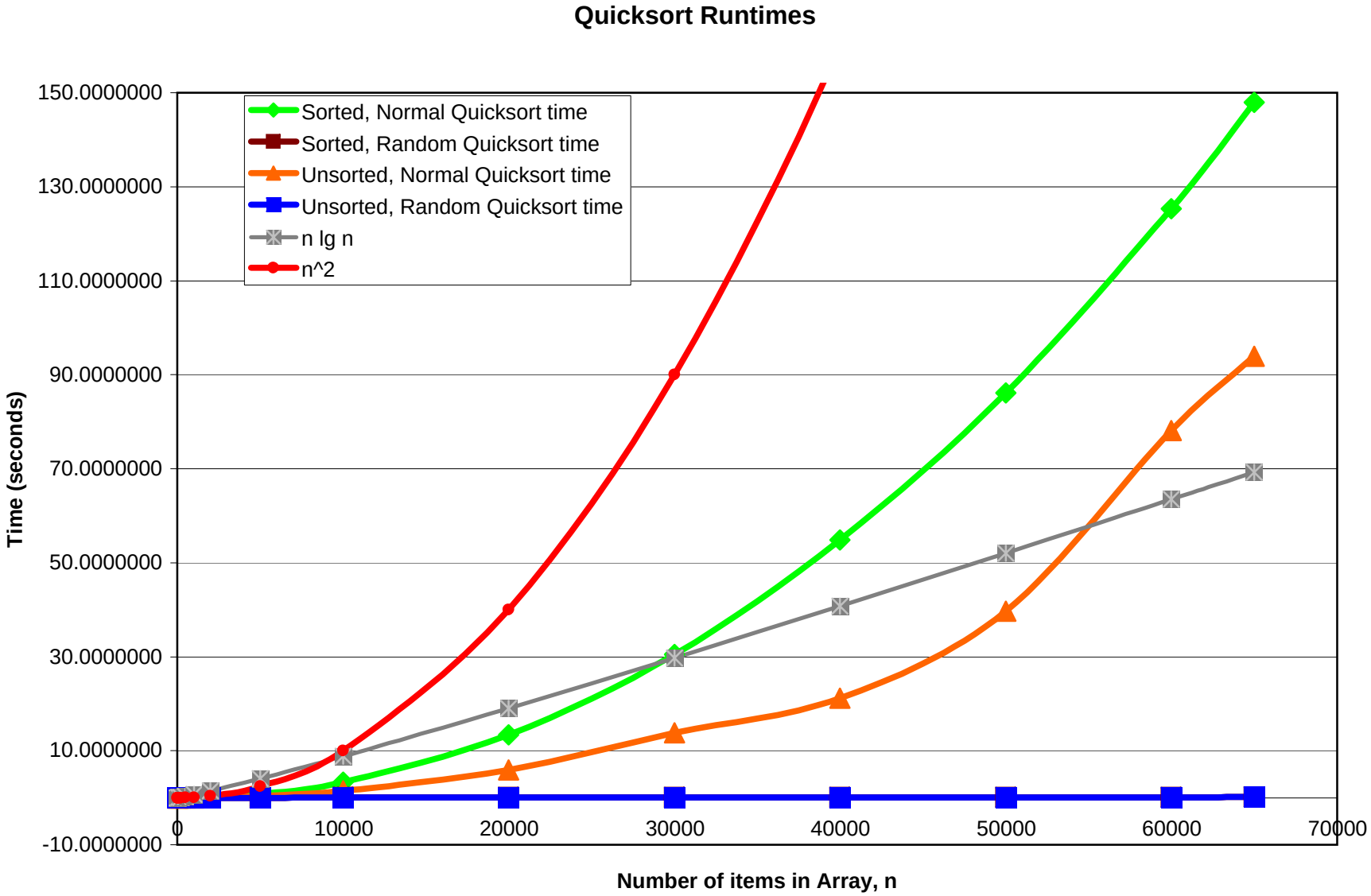
While the orange line does dip below and jump above the $n \lg n$ line, the overall trend of the line falls upon an $n \lg n$ slope, asymptotically speaking. Therefore it seems to be bound by an $n \lg n$ function, and so does indeed have a $\Theta(n \lg n)$ runtime.

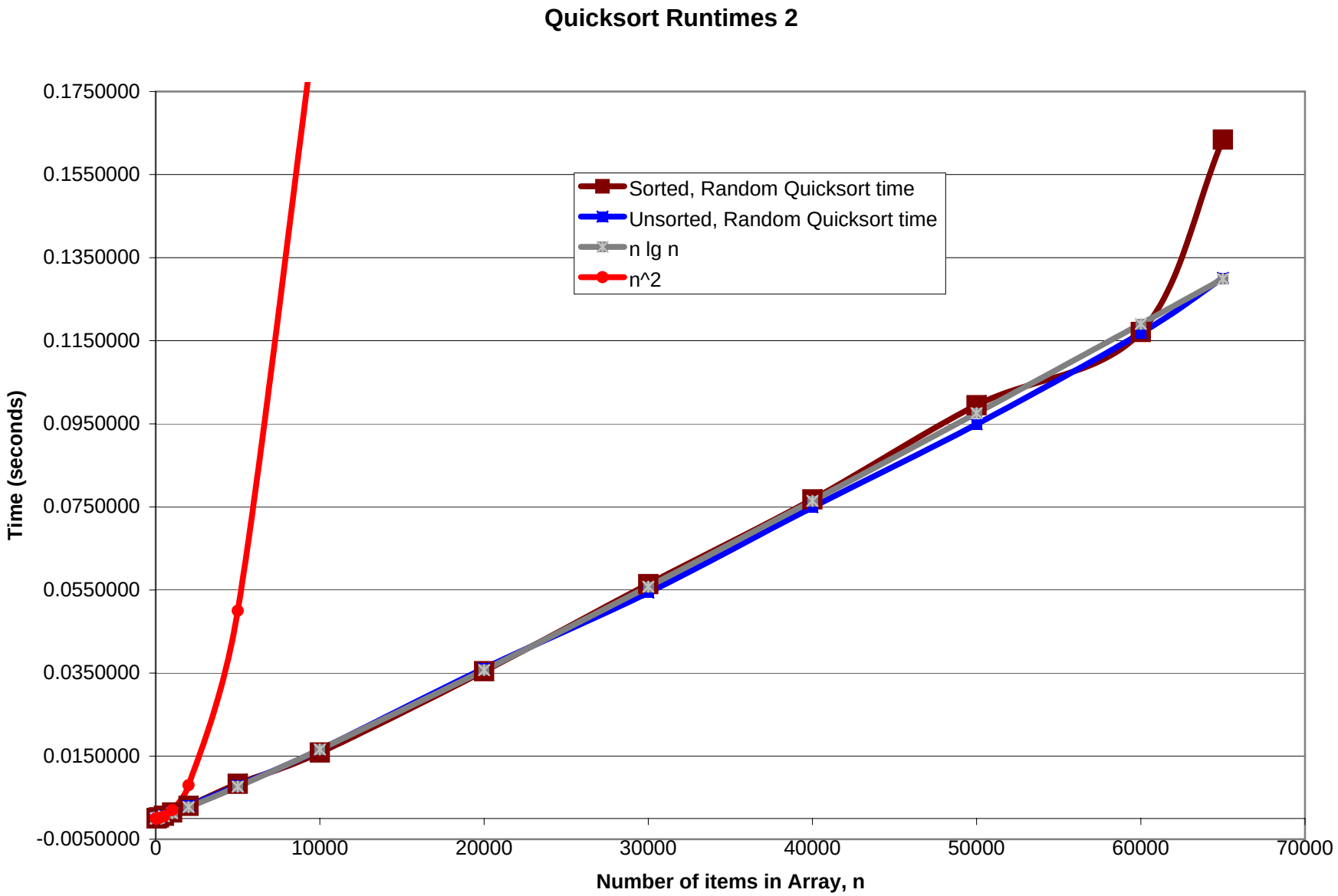
| Number of Items, n | Sorted, Normal Quicksort time | Sorted, Random Quicksort time | Unsorted, Normal Quicksort time | Unsorted, Random Quicksort time | n lg n | n^2 | n lg n | n^2 |
|---|---|---|---|---|---|---|---|---|
| 10 | 0.0000181 | 0.0000261 | 0.0000069 | 0.0000132 | 0.002215 | 0.00001 | 4.15241E-06 | 0.0000002 |
| 20 | 0.0000310 | 0.0000291 | 0.0000108 | 0.0000234 | 0.005763 | 0.00004 | 1.08048E-05 | 0.0000008 |
| 50 | 0.0001164 | 0.0000683 | 0.0000474 | 0.0000610 | 0.018813 | 0.00025 | 3.52741E-05 | 0.000005 |
| 100 | 0.0003900 | 0.0000961 | 0.0002786 | 0.0001374 | 0.044292 | 0.001 | 8.30482E-05 | 0.00002 |
| 200 | 0.0014104 | 0.0002679 | 0.0006694 | 0.0002612 | 0.101918 | 0.004 | 0.000191096 | 0.00008 |
| 500 | 0.0083398 | 0.0006511 | 0.0041827 | 0.0006533 | 0.298859 | 0.025 | 0.000560362 | 0.0005 |
| 1000 | 0.0329680 | 0.0014263 | 0.0120727 | 0.0014078 | 0.664386 | 0.1 | 0.001245723 | 0.002 |
| 2000 | 0.1315803 | 0.0030348 | 0.0581354 | 0.0030867 | 1.462105 | 0.4 | 0.002741446 | 0.008 |
| 5000 | 0.8206673 | 0.0083424 | 0.3809680 | 0.0078953 | 4.095904 | 2.5 | 0.00767982 | 0.05 |
| 10000 | 3.3113933 | 0.0158652 | 1.5053233 | 0.0165709 | 8.858475 | 10 | 0.01660964 | 0.2 |
| 20000 | 13.3521667 | 0.0354580 | 5.8641100 | 0.0361301 | 19.05028 | 40 | 0.035719281 | 0.8 |
| 30000 | 30.5210333 | 0.0563976 | 13.7677400 | 0.0543761 | 29.74535 | 90 | 0.055772531 | 1.8 |
| 40000 | 54.8530000 | 0.0768084 | 21.1882667 | 0.0749036 | 40.76723 | 160 | 0.076438562 | 3.2 |
| 50000 | 86.1390333 | 0.0994891 | 39.7349000 | 0.0948944 | 52.03213 | 250 | 0.097560253 | 5 |
| 60000 | 125.3110000 | 0.1171000 | 78.0729667 | 0.1168480 | 63.4907 | 360 | 0.119045062 | 7.2 |
| 65000 | 147.9730000 | 0.1634257 | 93.9305000 | 0.1301340 | 69.28199 | 422.5 | 0.129903736 | 8.45 |

| number of items, n | Sorted, Normal Quicksort time | Sorted, Random Quicksort time | Unsorted, Normal Quicksort time | Unsorted, Random Quicksort time |
|---|---|---|---|---|
| 10 | 0.0000192 | 0.0000303 | 0.0000069 | 0.0000136 |
| 20 | 0.0000343 | 0.0000269 | 0.0000116 | 0.0000251 |
| 50 | 0.0001173 | 0.0000634 | 0.0000406 | 0.0000558 |
| 100 | 0.0003903 | 0.0001451 | 0.0003050 | 0.0001710 |
| 200 | 0.0014184 | 0.0002679 | 0.0006351 | 0.0002678 |
| 500 | 0.0083387 | 0.0006386 | 0.0072864 | 0.0006706 |
| 1000 | 0.0328744 | 0.0014416 | 0.0135911 | 0.0014030 |
| 2000 | 0.1313740 | 0.0030734 | 0.0373893 | 0.0030775 |
| 5000 | 0.8232090 | 0.0086139 | 0.2194990 | 0.0079036 |
| 10000 | 3.3330100 | 0.0138723 | 1.6433900 | 0.0158507 |
| 20000 | 13.3481000 | 0.0398624 | 8.1143600 | 0.0335242 |
| 30000 | 30.5197000 | 0.0567847 | 7.7829200 | 0.0525154 |
| 40000 | 54.8883000 | 0.0789600 | 15.1158000 | 0.0746850 |
| 50000 | 86.1740000 | 0.0979216 | 42.6456000 | 0.0960548 |
| 60000 | 125.1620000 | 0.1128310 | 57.9393000 | 0.1207930 |
| 65000 | 148.1100000 | 0.1267020 | 61.0580000 | 0.1290070 |

| number of items, n | Sorted, Normal Quicksort time | Sorted, Random Quicksort time | Unsorted, Normal Quicksort time | Unsorted, Random Quicksort time |
|---|---|---|---|---|
| 10 | 0.0000200 | 0.0000318 | 0.0000069 | 0.0000124 |
| 20 | 0.0000299 | 0.0000275 | 0.0000101 | 0.0000230 |
| 50 | 0.0001165 | 0.0000823 | 0.0000910 | 0.0000666 |
| 100 | 0.0003870 | 0.0000120 | 0.0003127 | 0.0001172 |
| 200 | 0.0014031 | 0.0002782 | 0.0004950 | 0.0002675 |
| 500 | 0.0085814 | 0.0006232 | 0.0037757 | 0.0006280 |
| 1000 | 0.0328594 | 0.0013745 | 0.0129723 | 0.0013582 |
| 2000 | 0.1321670 | 0.0028683 | 0.0821879 | 0.0029141 |
| 5000 | 0.8195050 | 0.0085388 | 0.2471950 | 0.0079969 |
| 10000 | 3.2922900 | 0.0173227 | 1.7040500 | 0.0163255 |
| 20000 | 13.3401000 | 0.0309246 | 3.1954400 | 0.0378905 |
| 30000 | 30.4515000 | 0.0588950 | 20.0357000 | 0.0541627 |
| 40000 | 54.6792000 | 0.0760272 | 14.6107000 | 0.0733593 |
| 50000 | 85.7798000 | 0.1063600 | 42.7903000 | 0.0934855 |
| 60000 | 125.7230000 | 0.1148180 | 94.6022000 | 0.1113470 |
| 65000 | 146.8470000 | 0.1316130 | 129.4660000 | 0.1334590 |

| number of items, n | Sorted, Normal Quicksort time | Sorted, Random Quicksort time | Unsorted, Normal Quicksort time | Unsorted, Random Quicksort time |
|---|---|---|---|---|
| 10 | 0.0000150 | 0.0000162 | 0.0000069 | 0.0000137 |
| 20 | 0.0000289 | 0.0000330 | 0.0000107 | 0.0000220 |
| 50 | 0.0001154 | 0.0000593 | 0.0000106 | 0.0000607 |
| 100 | 0.0003926 | 0.0001311 | 0.0002180 | 0.0001240 |
| 200 | 0.0014096 | 0.0002575 | 0.0008782 | 0.0002484 |
| 500 | 0.0080992 | 0.0006916 | 0.0014861 | 0.0006614 |
| 1000 | 0.0331702 | 0.0014628 | 0.0096546 | 0.0014623 |
| 2000 | 0.1312000 | 0.0031627 | 0.0548290 | 0.0032685 |
| 5000 | 0.8192880 | 0.0078744 | 0.6762100 | 0.0077854 |
| 10000 | 3.3088800 | 0.0164007 | 1.1685300 | 0.0175364 |
| 20000 | 13.3683000 | 0.0355871 | 6.2825300 | 0.0369756 |
| 30000 | 30.5919000 | 0.0535132 | 13.4846000 | 0.0564503 |
| 40000 | 54.9915000 | 0.0754380 | 33.8383000 | 0.0766664 |
| 50000 | 86.4633000 | 0.0941857 | 33.7688000 | 0.0951430 |
| 60000 | 125.0480000 | 0.1236510 | 81.6774000 | 0.1184040 |
| 65000 | 148.9620000 | 0.2319620 | 91.2675000 | 0.1279360 |


Quicksort Runtimes
150.0000000
130.0000000
110.0000000
90.0000000
70.0000000
50.0000000
30.0000000
10.0000000
-10.0000000
Time (seconds)
0
10000
20000
30000
40000
50000
60000
70000
Number of items in Array, n
Sorted, Normal Quicksort time
Sorted, Random Quicksort time
Unsorted, Normal Quicksort time
Unsorted, Random Quicksort time
n lg n
n^2


Quicksort Runtimes 2
Time (seconds)
0.1750000
0.1550000
0.1350000
0.1150000
0.0950000
0.0750000
0.0550000
0.0350000
0.0150000
-0.0050000
0
10000
20000
30000
40000
50000
60000
70000
Number of items in Array, n
Sorted, Random Quicksort time
Unsorted, Random Quicksort time
n lg n
n^2

***Sources:***


**1) Randomized Algorithms, Randomized Quicksort, Probability.**

http://www.cs.rpi.edu/~yener/TEACHING/Algorithms/documents/randomize.pdf

(found via Google - http://www.google.com/search?sourceid=navclient&ie=UTF-8&oe=UTF-8&q=randomized+quicksort)