Sorting Algorithm Analysis

University of Maryland University College

By:

Paul Minniti

**Introduction**

Quicksort is an algorithm for sorting a list or an array of things. In most cases it is more efficient than the other sorting algorithms. In order to understand how to program quicksort, you must know that the pivot is the last element of an array or list, the lowest index or wall and the left most element or the current element. It will look at all of the elements minus the pivot and put all of the elements smaller than the pivot to the left of the wall. After its first pass it will drop the pivot on the wall or between all the numbers small and larger than it. This process will continue until the list or array is in the correct order.At worst quicksort is an O(n^2) if the pivot happens to be the highest or lowest value but typically performs like an O(n log n) algorithm.

**Critical Operations**

**Recursion**

Find the results of a problem by breaking it down into smaller and smaller pieces until you find a solution.

**Iteration**

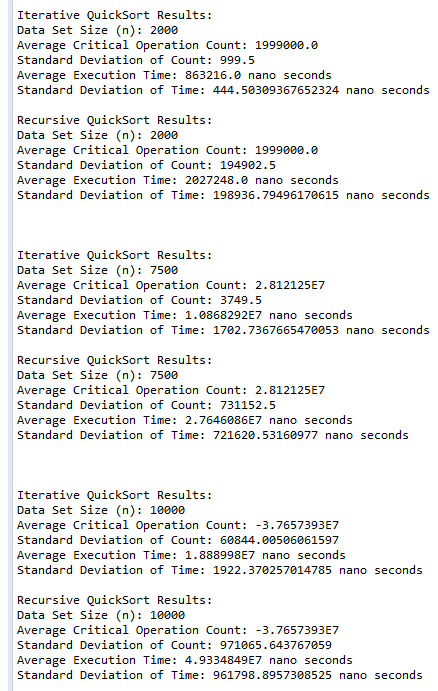
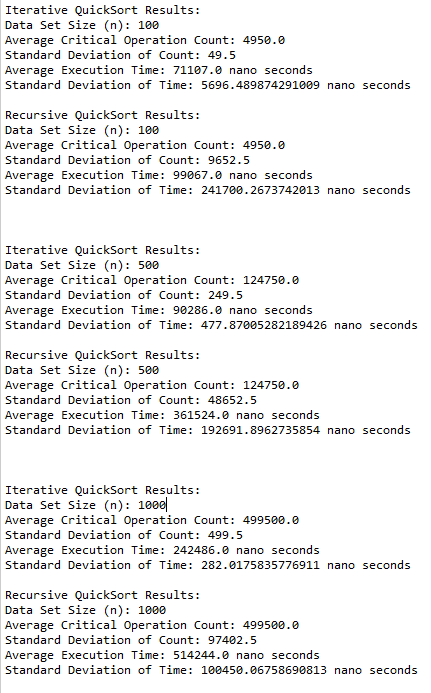
Iterate or repeat a problem until the loop counter reaches its limit.

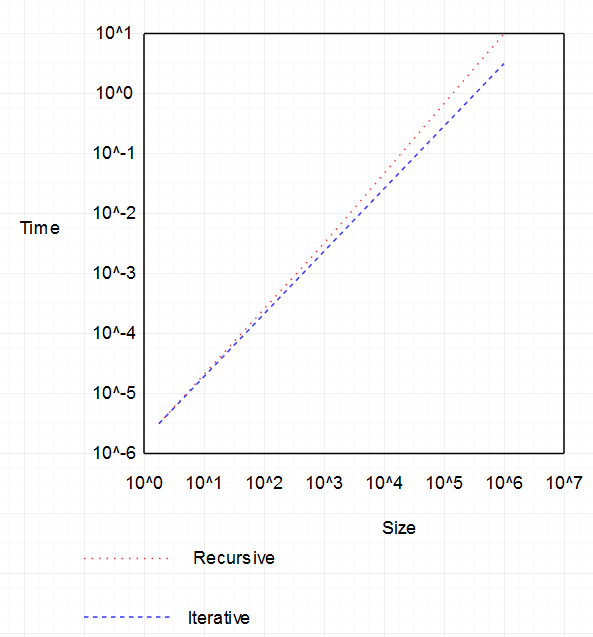
Both the recursive and iterative version of quicksort solve a task one piece at a time until a result is found.

**Big-O Analysis**

Best case running time occurs when the elements on both sides of the wall are balanced with n/2 elements or in the case of an odd size within one value from one another. Using big-O notation we get O(n log n). Worst case running time occurs if the list or array has an odd number of elements and both sides of the wall have an unbalanced portion of elements. For some constant, on (n-1) the recursice call takes c(n-1) time. Using big-O notation we get O(n^2).Average case running time occurs if at worst one side gets 3n/4 and the other gets n/4. Therefore because it cannot be better than the best case using big-O notation we get O(n log n).

**Results Analysis**

****



Time Complexity

|  |  |  |
| --- | --- | --- |
| Best Case | Average Case | Worst Case |
| O(n log n) | O(n log n) | O(n^2) |

Execution Times

|  |  |  |
| --- | --- | --- |
| Size | Recursive Time | Iterative Time |
| 100 | 0.000099067 seconds | 0.000071107 seconds |
| 500 | 0.000361524 seconds | 0.000090286 seconds |
| 1000 | 0.000514244 seconds | 0.000242486 seconds |
| 2000 | 0.002027248 seconds | 0.000863216 seconds |
| 7500 | 0.027646086 seconds | 0.010868292 seconds |
| 10000 | 0.049334849 seconds | 0.018889980 seconds |

Since the average case time complexity is O(n log n)

Therefore ,Time required= O(n log n)\*Constant

Assuming that constant to be k and calculating it for each case

For recursive case

for n=100.0 value of k for recursive approach= 2.1512125719354726E-7

for n=500.0 value of k for recursive approach= 1.1634651591040351E-7

for n=1000.0 value of k for recursive approach= 7.444444385061862E-8

for n=2000.0 value of k for recursive approach= 1.333557436627158E-7

for n=7500.0 value of k for recursive approach= 4.131218159723185E-7

for n=10000.0 value of k for recursive approach= 5.35646317155754E-7

Standard deviation of k=0.06927226569335379

For iterative case

for n=100.0 value of k for iterative approach= 1.544068886234726E-7

for n=500.0 value of k for iterative approach= 2.9056055851027015E-8

for n=1000.0 value of k for iterative approach= 3.510344391293065E-8

for n=2000.0 value of k for iterative approach= 5.678378354377702E-8

for n=7500.0 value of k for iterative approach= 1.624073848123536E-7

for n=10000.0 value of k for iterative approach= 2.0509535193156972E-7

Standard deviation of k= 0.026851635128443966

So, we observe that value of k is fairly constant and has a small standard deviation, therefore the theoretical analysis is confirmed by practical results, and we also observe it is more pronounced in iterative case.

For standard deviation of Execution time of the algorithm we observe that in both recursive and iterative approach the standard deviation First decreases and then increases, therefore it is very sensitive for very large and very small data.

Possible reason for this behavior is that is data if very small then the number of instructions executed will be limited and for different data set of same size some take best case complexity and other will take worst case complexity and thus, the standard deviation will be huge, and for very large data set mostly, there will be an average case complexity of n(logn), because as the data set increases the possibility of variation decreases and algorithm tends to shift towards lesser variation, or asymptotically leading towards average case complexity. For very large data set however the reason of very large runtime deviation could be garbage collection eventually has to do much more work, which increases the number of outliers