On How Statistics Provides a Reliable and Valid Measure for an Algorithm's Complexity

*Soubhik Chakraborty**
*Suman Kumar Sourabh****
Basant Kumar Das
University Department of Statistics and Computer Applications
T.M. Bhagalpur University
Bhagalpur – 812007
I N D I A
**email:soubhikc@yahoo.co.in*
****email:suman_sourabh2003@yahoo.com*

*Kiran Kumar Sundararajan***
Department of Statistics
Bharathiar University
Coimbatore - 641046
I N D I A
*** email:kkmars@rediffmail.com*

Abstract: Stochastic modelling of deterministic **computer experiments** was strongly and correctly advocated by Prof. Jerome Sacks and others [see J. Sacks, W. Welch, T. Mitchell and H. Wynn, Design and Analysis of Computer Experiments, Statistical Science, vol. 4. No. 4, 1989] in order to reduce the cost of prediction, quantifying the amount of accuracy sacrificed in the bargain. Following some of these guidelines, the authors in the present manuscript have made an interesting statistical adventure by empirically deriving the difficult O(nlogn) average case complexity of the popular Quicksort algorithm. Since the strategy could be easily applicable, with little or no modification, to any arbitrary algorithm of similar or nearly similar complexity and more importantly where a formal theoretical proof might be found wanting, [see also the first chapter in the book Data Structures and Algorithms by A. Aho, J. Hopcroft, J. Ullman, Addison Wesley], the authors propose Statistics as a reliable and valid tool for measuring an arbitrary algorithm's time complexity. As an added attraction, the paper further shows how the sorting efficiency of Quicksort is affected by input from specific distributions, citing the Binomial case, indicating the motivation of further research in that direction in the future, especially covering mixture distributions.
[note: a **computer experiment** is a series of runs of a code for various inputs; it is called deterministic if feeding the same inputs leads to identical observations]

Key words: Quicksort(nonrecursive) algorithm, the big-O, time complexity, applied regression analysis

Section I gives the introduction, section II(A) gives the algorithm description and II(B) a simple QBASIC program on Quicksort(nonrecursive), section III gives the statistical analysis, while section IV gives conclusion and future work.

Section I

Introduction: The Quicksort algorithm given by Prof. C. A. R. Hoare[1] is probably the most widely used than any other (internal) sorting algorithm. It is popular because it is not difficult to implement, it is a good general purpose sort(i.e., works well in a variety of situations), and consumes less resources than any other sorting algorithm. The desirable features of this algorithm are that it is in-place(i.e., uses a small auxiliary stack), requires only about $n\log_2 n$ operations on the average to sort n items and has an extremely short inner loop. The drawbacks of the algorithm are that it is recursive (implementation is complicated if recursion is not available or not wanted, to avoid the tremendous demands recursion calls for from the system), has a bad worst case where it takes about n^2 operations and is fragile; a simple mistake may go unnoticed and could cause it to perform badly for some inputs. See also Sedgewick [2], Aho, Hopcroft, Ullman[12].

Almost from the moment Hoare first published the algorithm[1], improved versions have been appearing in the literature. Many ideas have been tried and analyzed
but it is easy to be deceived, because the algorithm is so well balanced that the effects of improvements in one part of the program can be more than offset by the effects of bad performance in another[2]. Here we will be subjecting a nonrecursive version of this algorithm to a careful statistical analysis.

Section II(A)

Mechanism of the Quicksort algorithm: The Quicksort method separates a set of numbers into a partition or interval containing larger numbers and another partition containing relatively smaller numbers. Then the process is repeated with these new partitions (*the divide and conquer strategy*!).To understand the mechanism, let us think of sorting some numbers which are written on a line from left to right. As an example, we may wish to sort the following numbers:

19 29 25 17 22 18 20

into an increasing order from left to right (the sorted order would obviously be 17, 18, 19, 20, 22, 25, 29). We will describe and use an interval partition scheme which will be used over and over to accomplish the sort. This scheme will be written as a subroutine in in the main Quicksort program.

The interval partition scheme

Consider the set of unsorted numbers 19, 29, 25, 17, 22, 18, 20. Let us assume these numbers are in the subscripted variables A(1), A(2)….A(7). It is important to know
in this scheme that the subscripts for this interval of numbers range from I=1 on the
left to I=7 on the right[for convenience, I am confining the reader to the example only
which will be generalized in the program implementation in section II(B)]. Take a copy

of the leftmost number, 19=A(1), and put it on a “pedestal X”i.e., set X=A(1) so that we can clearly see it and use it for comparison.

19
X

19	29	25	17	22	18	20
A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)

Now start with the rightmost number , 20=A(7) and compare X with A(7). This first number, 20, is larger than 19, so leave it where it is. We have,

19   29   25   17   22   18   20

Move down to 18=A(6) and compare with X. This one is smaller, so move it down to A(1) and let it be inserted into A(1). Remember this will destroy 19 but that we have a copy of 19 in X. We have,

18   29   25   17   22   18   20

Now go up to A(2) and compare 29 with 19 in X. This one is larger, so move a copy of it up to A(6). We have,

18   29   25   17   22   29   20

Now go down to A(5) and compare 22 with 19. It is larger, so leave it. We have,

18   29   25   17   22   29   20

Go down to A(4) and compare 17 with 19. It is smaller, so move it down to A(2).

18   17   25   17   22   29   20

Now go up to A(3) and compare 25 with 19.Being larger, 25 moves up to A(4).
This gives

18   17   25   25   22   29   20

Now go down to A(3), but discover that we have already reached position 3 coming up from below. So stop the comparisons and place 19 into this position. We have,

```
 18     17     19     25     22     29     20
A(1)   A(2)   A(3)   A(4)   A(5)   A(6)   A(7)
```

Now that we have concluded the first application of this interval separation scheme, we have reached our intermediate goal. All of the numbers to the right of 19 = A(3) are larger than 19, and all of the numbers to the left of 19 are smaller than 19. This number 19 has found its permanent place but the other numbers have not. So the numbers to the left become a new interval for the application of this scheme. Also, the numbers to the right form an interval ready for a repeat of this scheme.

In anticipation of applying this scheme to each interval mentioned above, we should not only conclude by placing 19 into its proper place, but we should also give the subscript ranges of the left and right intervals above. For the left interval, the subscript range is left=1 and right=2 and for the right interval, the range is left=4 and right=7. *To use this interval-separation scheme on every new interval until there are no intervals left is to sort all of our numbers.*

As we saw in the above application of the scheme, we start with one interval of numbers and usually end with two intervals of numbers. Since only one interval of numbers can be worked with at a time, the left and right subscripts of the other interval must be saved for later. This saving will be done on a *stack* consisting of a two dimensional array called S(s in the program). The left subscript will be saved in column 1 and the right subscript will be saved in column 2.As the total process goes on, there may be several intervals waiting on the stack for processing. In the very beginning we might as well put the first interval,1 and N(n in the program), on the stack and start using it. Then *when the stack is empty, the sorting is done.*

There are some special cases in dealing with these intervals. For some interval of numbers it may happen that the leftmost number, X(x in the program), is already the smallest of the set. Then at the end of the scheme we should make the left and right subscripts for the left interval zero to signal that there is no left interval. Again, for some interval of numbers it may happen that the left interval below X contains only one number. This is a nice special case in that both this small number and X are in their permanent positions for the sort. So again there is no need for a left interval, and the left and right subscripts for it should be set to zero. These special cases for the left interval may happen for the right interval. A similar use of zero for left and right subscripts would also signal these special cases on the right.

As this is a difficult program to implement, especially without recursion, I am providing for the benefit of the reader a simple QBASIC program(non - recursive; BASIC does not support recursion; there used to be a version of BASIC called BBC BASIC which probably supported recursion but is obsolete now; the other reason for preferring the non-recursive version has been explained earlier). Thus on one hand the reader will have a program which runs on a compiler, on the other a program that is easy to follow, since the syntax and semantics of BASIC are simpler than those of any other computer language. Run time will be calculated using the usual TIMER function. The program is given in section II(B). Array a is filled with n independent discrete uniform variates in the range 1,2,3…n which are then quicksorted.

Section II(B)

```
REM quicksort(non-recursive)
CLS
INPUT "how many numbers to sort"; n
DIM a(n), r(n), s(n, 2)
RANDOMIZE TIMER
FOR i = 1 TO n: a(i) = INT(n * RND) + 1: NEXT i
GOTO 20
10  GOTO 160
20 st = TIMER
z = 1
s(z, 1) = 1
s(z, 2) = n
30 IF s(z, 1) = 0 THEN GOTO 10
l1 = s(z, 1)
r1 = s(z, 2)
s(z, 1) = 0: s(z, 2) = 0
z = z - 1
40 GOSUB 70
IF l2 = 0 AND l3 = 0 THEN GOTO 30
IF l2 <> 0 AND l3 = 0 THEN GOTO 60
IF l2 = 0 AND l3 <> 0 THEN GOTO 50
z = z + 1
s(z, 1) = l2
s(z, 2) = r2
50 l1 = l3
r1 = r3
GOTO 40
60 l1 = l2
r1 = r2
GOTO 40
70 x = a(l1)'subroutine begins from here'
r = r1
i = l1
80 FOR j = r TO l1 STEP -1
IF i = j THEN GOTO 110
IF x > a(j) THEN GOTO 90
NEXT j
90 a(i) = a(j)
l = i + 1
FOR i = l TO r1
IF i = j THEN GOTO 110
IF x < a(i) THEN GOTO 100
NEXT i
100 a(j) = a(i)
r = j - 1
```

```
GOTO 80
110 a(i) = x
IF i <= l1 + 1 THEN GOTO 120
l2 = l1
r2 = i - 1
GOTO 130
120 l2 = 0
r2 = 0
130 IF j >= r1 - 1 THEN GOTO 140
l3 = j + 1
r3 = r1
GOTO 150
140 l3 = 0
r3 = 0
150 RETURN
160 et = TIMER
PRINT "run time in sec="; et - st
END
```

*[*As printing the array before and after sorting is optional, we have not included it in the program. QBASIC does not require line numbers and therefore we have avoided unnecessary line numbers except when a reference is required to transfer the control of the program.]

Section III

The following results were obtained for 3 trials:-

Table 1

	y(run time in sec)			
n	trial 1	trial2	trial3	mean
500	0.1640625	0.171875	0.15625	0.1640625
1000	0.390625	0.375	0.3828185	0.3828185
1500	0.609375	0.65625	0.5390625	0.6015625
2000	0.828125	0.8828125	0.875	0.8619792
2500	1.085938	1.046875	1.101563	1.078125
3000	1.3125	1.304688	1.382813	1.3333334
3500	1.765625	1.585938	1.640625	1.664063
4000	1.976563	1.804688	1.875	1.885417
4500	2.140625	2.203125	2.023438	2.122396
5000	2.703125	2.359375	2.257813	2.440104

In what follows, we shall write y to mean the mean run time taken over three trials, as a function of n. First, let us show that we can also derive experimentally what we already know from theory. Define x= $n\log_2 n$ and, since the values of x will not be equally spaced even if n is equally spaced, we shall form a divided difference table(table 2):-

Table 2

$x = n\log_2 n$	y	first order divided difference $(y_1-y_0)/(x_1-x_0)$ etc.
4482.892	0.1640625	
		3.989683 E-5
9965.784	0.3828125	
		3.732721 E-5
15826.12	0.6015625	
		4.265317 E-5
21931.57	0.8619792	
		3.43759 E-5
28219.28	1.078125	
		3.967209 E-5
34652.24	1.333334	
		5.046412 E-5
41205.99	1.664063	
		3.325057 E-5
47863.14	1.885417	
		3.512072 E-5
54610.69	2.122396	
		4.710319 E-5
61438.56	2.440104	

Fundamental theorem of divided difference: The n-th divided difference of a polynomial of degree n is constant and higher divided differences are zero.[5]
Conversely, if the n-th divided difference of a tabulated function is the first to be constant and the (n+1)th divided difference the first to be zero, the function is a polynomial of degree n. Since a portion of the run time is also affected by the system environment, we do not expect an exact constancy. However, from the divided difference table above, we may safely take y as a linear function of x whence $y = a + bx$. This implies $y=O(x)$.But we had taken $x=n\log_2 n$ and hence $y=O(n\log_2 n)$. The same result can be derived from purely theoretical considerations also. See, for example, Knuth[3]. The reader may put an intelligent question here: What is so special about the experimental derivation? Well, a number of things. First, recall that **a measure is called reliable if it is able to measure what it is supposed to while it is termed as valid if it produces results which matches** those accepted as standard**. The very fact that we are able to measure the time complexity of the difficult but useful quicksort algorithm experimentally proves that the statistical approach is reliable while the matching of the empirical and theoretical results endorses the validity of the approach. Secondly, we can, and in fact will, provide numerical values of the constants a and b in our system which is impossible to derive theoretically. These constants are of paramount importance to the statistician for a number of reasons, for example, in predicting the run time without actually running the program especially for a very large n for which the program breaks down with an "out of memory" message; the statistician, having estimated the numerical values of a and b, will simply substitute the

value of n in the equation y= a + bnlog$_2$n and predict the run time precisely in seconds, thus leading you to a world of fantasy where your system does not permit you to reach physically! Again, if there is a competing sorting algorithm with the same order of complexity as the quicksort, it is the constants which provides us with valuable information to compare and assess which is the better of the two when it comes to performance.The argument holds for "close" algorithms also[14]. Regressing y on x linearly, one gets

y predicted= -0.0203 +0.000040 x; x=nlog$_2$n

The standard errors of the least square estimates of a and b are respectively 0.01535 and 0.00000042. The ANOVA table for regression analysis is as follows:-

Table 3

Source	**df**	**ss**	**ms**	**F**	**Remarks**
Regression	**1**	**5.2867**	**5.2867**	**9194.261**	**Highly significant**
Residual error	**8**	**0.0046**	**0.000575**		**compared to table**
Total	**9**	**5.2913**			**F at 1,8 df**
					(take as 5% or1%)

R^2 = 5.2867/5.2913=0.9991307. Hence 100R^2%(coefficient of determination)=99.91% of variation in the response variable is explained by the predictor.

Using a MINITAB statistical package, we also obtained a number of useful information such as predicted y, standard error of predicted y, residuals, standardised residuals, plots(normal probability, residual), COOK's distance, DFITs, etc. Due to shortage of space, we are summarising some of these results in table 4:-

---

**in case of a contradiction[11], we would expect that the statistical approach should not lead to statements mathematically weaker than those theoretically derivable.

Table 4

Observed y(rounded)	predicted y	S.E.(pred. y)	residual	std. res.
0.16406	0.15805	0.01376	0.00602	0.30
0.38281	0.37614	0.01193	0.00667	0.32
0.60156	0.60925	0.01017	-0.00769	-0.35
0.86198	0.85211	0.00869	0.00987	0.44
1.07813	1.10222	0.00777	-0.02409	-1.06
1.33333	1.35810	0.00767	-0.02477	-1.09
1.66406	1.61879	0.00852	0.04527	2.01R*
1.88542	1.88360	0.01007	0.00182	0.08
2.12240	2.15200	0.01210	-0.02960	-1.42
2.44040	2.42359	0.01442	0.01651	0.86

R* denotes an observation with a high standardized residual.

Another strength of a statistical approach is that it can simulate the input for some specific probability distribution characterizing the sorting elements and then study its effect on the sorting efficiency. For example, for the Binomial B(m,p) distribution input, we have some very interesting results on the same quicksort algorithm as follows:-

[Hint: Binomial(m,p) input may be obtained by incorporating the following subprogram :

```
INPUT m,p
FOR i=1 TO n
xx=0
FOR j=1 TO m
t= RND:IF t<p THEN xx=xx+1
NEXT j
a(i)=xx
NEXT i
```

See also Kennedy and Gentle[6] for how to get other distributions input]. For a theoretical discussion, see Hosam M. Mahmoud, Sorting :a Distribution Theory, John Wiley & Sons, 2000.

Table 5

Binomial(m,p) distribution input

(Approximate run time in sec upto two places of decimals)

	p= 0.2			p= 0.5			p= 0.8		
	m=10	m=100	m=1000	m=10	m=100	m=1000	m=10	m=100	m=1000
n									
1000	2.59	0.88	0.55	1.49	0.77	0.49	1.98	0.88	0.49
2000	8.74	3.13	1.54	7.36	2.31	1.32	9.94	2.96	1.43
3000	19.61	6.54	2.63	12.85	5.38	2.31	17.63	6.32	2.64

We make the following observations from table 5:-

(i) For a given n and p, quicksort time decreases noticeably as the parameter m in Binomial(m,p) increases. Note that an increase in m leads to an increase in the range [0,1,2,…m] of the sorting elements.

(ii) For a given n and m, quicksort time is least for p=0.5 and increases both for p=0.2 (<0.5) and p=0.8(>0.5)

In conclusion, quicksort is more efficient for p=0.5 and a large m in case of a Binomial(m,p) input.

We will settle one final question before closing this manuscript. How did the statistician know that the logarithm will have a base 2? Suppose it has a base e. See that $\log_2 n = \log_e n \log_2 e = c\log_e n$ where $c = \log_2 e$, a constant whence $O(n\log_2 n)$ $= O(cn\log_e n) = O(n\log_e n)$, by definition of the big-O[see Knuth[9]]. Take $z = n\log_e n$ and regress y on z. The b-estimate will change and numerically equal c times the earlier estimate, so that predicted y is unaffected. Q. E. D.!

Do not forget, above all, that the statistician has the satisfaction of working on actual run time(which we believe is something to be observed and recorded and not merely something to be predicted from the desk!) and that the program run time is nothing but *a weighted sum of the computing operations*, the weights being the corresponding run time each consumes. Don't you feel that this approach is more *realistic* than merely enumerating the computing operations?[see[7]].

Section IV
Conclusion and future work

We have established, through our analysis of the famous quicksort, that the statistical approach for complexity analysis of computer algorithms is both reliable and valid with a number of other dividends when it comes to estimating the constants associated with the big – O, prediction beyond physical reach and in assessing the algorithm's efficiency by simulating from different probability distributions as input elements .We propose to make similar studies for other algorithms and other probability distributions, both discrete and continuous as well as mixtures.

Remark: Since O(nlogn) + O(n) = O(nlogn) [prove!] it might be worthwhile to work with a model y= a + bnlogn + cn + ε which is also a linear model of the type $y = a + bX_1 + cX_2 + \varepsilon$. The reader may note that, to the extent reasonable, the authors did not make use of the theoretical counterpart and yet came close to it arguing purely empirically!

References:-

[1] C.A.R. Hoare, Quicksort, the Computer Journal, vol.5, no.1, 1962

[2] R. Sedgewick Algorithms, Chap.9, Addison Wesley

[3] D.E. Knuth, Sorting and Searching(The art of computer programming vol.3), Addison-Wesley

[4] N. Draper and Harry smith, Applied Regression Analysis, John Wiley & Sons

[5] J. Scarborough, Numerical Mathematical Analysis, the Johns Hopkins Press

[6] W. Kennedy and J. Gentle, Statistical Computing, Marcel Dekker Inc

[7] S. Chakraborty and P.P. Choudhury, A Statistical Analysis of an Algorithm's Complexity, Applied Mathematics Letters, vol.13, no. 5, 2000 (abstract pub. in Mathematical Reviews, Amer. Math. Soc., Dec. 2000)

[8] Robert G. Thompson, BASIC a modular approach, chap. 11, CBS pub & dist(Ind)

[9] D.E. Knuth, The Art of Computer Programming, vol.1, Addison Wesley

[10] D.E. Knuth, The Art of Computer Programming, vol.2, Addison Wesley

[11] S. Chakraborty and P.P. Choudhury, Can Statistics Provide a Realistic Measure for an Algorithm's Complexity?, Applied Mathematics Letters, 12(7), 1999

[12] A. Aho, J. Hopcroft, J. Ullman, Data Structures and Algorithms, Addison Wesley

[13] J. Sacks, W. Welch, T. Mitchell, H. Wynn, Design and Analysis of Computer Experiments, Statistical Science, vol. 4, No. 4, 1989

[14] S. Chakraborty, A. Mukherjee, P. Mukherjee, P. P. Choudhury, Is Replacement Sort Faster Than Bubble Sort?, Fifth Triennial International Calcutta Symposium, Dec 28-31, 2003, Kolkata, India