EVALUATING THE EFFICIENCY OF SORTING ALGORITHMS

Steve Legenhausen*

Research Assistant
University Computer Center
University of Minnesota

UCC Technical Report No. 2
September, 1971

*Present address: Department of Computer, Information, and
Control Sciences
University of Minnesota
EVALUATING THE EFFICIENCY OF SORTING ALGORITHMS

Steve Legenhausen*

Research Assistant
University Computer Center
University of Minnesota

UCC Technical Report No. 2

September, 1971

*Present address: Department of Computer, Information, and Control Sciences
University of Minnesota
The problem of determining the relative efficiencies of different sorting algorithms is discussed in this paper. General criteria for judging sorting performance are derived. Empirical tests for investigating sorting algorithms with respect to these criteria are given. These tests are applied to a study of Quicksort and Shellsort algorithms, and the superior performance of Quicksort is established.
I. Introduction

Sorting is an extremely useful operation in data processing. The number of data items in these applications is usually quite large, so it is important to have efficient sorting algorithms.

In this paper we consider only the restricted problem of sorting the elements of an array into ascending order. An unusually large variety of algorithms has been devised for this purpose, and finding the best ones can be a difficult problem. We further restrict our efforts to considering only general purpose sorting algorithms, such as would be suitable for incorporating in a program library. It is possible to construct very fast special purpose algorithms based on address calculation [9], but these suffer from the obvious defect that a different program must be written for each application. Most general purpose algorithms sort by comparing elements two at a time, and this is the only type of algorithm included in the present study.

Two very popular sorting methods, Quicksort and Shellsort, are described in section 2. The problem of evaluating the efficiency of sorting algorithms is taken up in section 3. Four general criteria for the performance of a sorting algorithm are derived, and current analytical studies are shown to be inadequate. Then in section 4 some empirical methods for investigating the relative efficiency of sorting algorithms are defined. These methods are applied to a study of the Quicksort and Shellsort algorithms.

2. Two Sorting Algorithms

2.1 Quicksort

The Quicksort method was conceived by C.A.R. Hoare [6,7]. The basic principle of the method is so simple and elegant that it can be described briefly. First, an element called the bound is chosen in some manner from the array, and the array is partitioned into two segments, the first consisting of elements which are less than or equal to the bound, and the other consisting of elements which are greater than or equal to the bound. The procedure is then applied recursively to the two seg-
ments of the array until the data are completely sorted.

Several suggestions for efficient implementations of the Quicksort method are also discussed by Hoare [7]. These suggestions have been followed up by various persons, and efficient algorithms have been described by Hibbard [4,5], Scowen [12], Singleton [14], and van Emden [16]. The success of the general Quicksort method provides a remarkable demonstration of the power of recursive algorithms.

The version of Quicksort developed by Singleton appears to be the fastest published sorting algorithm, based on the claims made in the literature. A Compass (Control Data 6000/7000 series assembly language) program based on Singleton's algorithm appears in Appendix A, and the performance characteristics of this program are studied in section 4. Analytical studies relating to some variations of Quicksort have appeared recently in the literature; see Frazer and McKellar [3], van Emden [15], and Hurwitz [8].

2.2 Shellsort

The Shellsort method, which first appeared in 1959, is named after D. L. Shell [13]. Although it is quite simple to implement Shellsort in a computer program, it is rather difficult to give a verbal description which does justice to the basic simplicity of the method. The reader who desires to learn something of the structure of the method should study the original flow chart given by Shell.

Minor improvements in the Shellsort method have been described by Frank and Lazarus [2] and Hibbard [5]. The algorithm developed by Hibbard is believed to be superior. Shellsort has been extremely popular since its inception and has been widely implemented. This popularity is due in part to the simplicity of the method combined with its relative efficiency.

A Compass program based loosely on Hibbard's algorithm is included in Appendix B. The performance characteristics of Shellsort are studied in section 4. The timing data given there were not obtained from the program shown in Appendix B but
rather from a similar although slightly more general implementation of Hibbard's algorithm. The running times of the two Shellsort programs are very close.

No satisfactory analysis of the expected running time of Shellsort has ever been published. Thus a comparison of the Quicksort and Shellsort algorithms is an example of considerable practical importance.

3. Evaluation of Sorting Algorithms

3.1 Analysis of Sorting Algorithms

With a rather large number of different sorting algorithms to choose from, finding the best one is a difficult problem. Perhaps the most satisfying way to approach this task would be by analyzing the characteristics of some different algorithms.

The most important facts to find out in the analysis of any algorithm are the expected running time and the amount of memory space required. It has become traditional to measure the running time of sorting algorithms by counting the average number of data comparisons made in sorting. Although this may be inadequate for the practical determination between algorithms, we can gain considerable insight into the problems of sorting by temporarily adopting this approach.

Every sorting algorithm must have an array of \( N \) elements to hold the data. In addition an ordinarily negligible amount of memory is required to hold the instructions. It is important to determine the amount of auxiliary storage space required beyond this, as this represents costly overhead.

3.2 Minimum Number of Comparisons

Perhaps the most interesting question to consider is that of determining the least average number of comparisons to sort \( N \) items which would be needed by any possible sorting algorithm. An answer to this question is known, provided we assume that the data consists of \( N \) unique, randomly ordered items, and is usually
given as

\[ \log_2 N! \]  

This interesting fact has been noticed by various authors, and it is usually proved by means of an information theoretic argument (see for example Hoare [7]). A discussion of this result, as well as a new proof based on combinatorial principles, is given by Morris [11].

We may estimate the magnitude of (1) with the aid of Stirling's approximation

\[ N! \approx \sqrt{2\pi N} \left(\frac{N}{e}\right)^N. \]

With a good slide rule we readily calculate

\[ \log_2 N! = N \log_2 N - 1.44N + 0.5 \log_2 N + 1.33 \]  

(2)

3.3 Evaluation Criteria

A brief consideration of the assumptions used to obtain (1) will prove instructive. In particular it should be noted that these assumptions have very little practical basis. It may often happen in sorting applications that the data contain many nonunique items, or that the items are not randomly ordered. Thus the efficiency of a sorting algorithm in these cases is of considerable practical interest.

We are now in a position to state criteria for a good sorting algorithm:

(a) Running time - Average running time for sorting \( N \) unique, randomly ordered items proportional to \( N \log_2 N \) (from (2)).

(b) Existing order - Improved running time for sorting nonrandomly ordered items.

(c) Equal items - Improved running time for sorting nonunique items.

(d) Memory space - Only a small constant amount of auxiliary memory space required.

These are extremely strong conditions, and many of the better known sorting methods lack in one or more areas. To illustrate briefly, it is well known that the
sorting method commonly referred to as "bubble sort" does not meet condition (a). The basis of this method is comparing every item to every other item. Thus to sort \( N \) items requires on the order of \( N^2 \) comparisons, and this method is too slow for practical use. The method based on merging is somewhat more successful. Each pass in this method consists of merging pairs of sorted segments obtained from the previous pass. To sort \( N \) items requires \( \log_2 N \) passes, and hence on the order of \( N \log_2 N \) comparisons. But merging also requires an auxiliary array of \( N \) elements, and the method fails to satisfy condition (d). In section 4 we investigate to what extent the conditions are satisfied by the Quicksort and Shellsort algorithms.

3.4 Difficulties in Evaluation

It may be quite difficult to analyze the performance of a sorting algorithm with respect to our criteria. In the first place estimating the average number of comparisons will not in general be sufficient for a practical determination of the running time of an algorithm. Other operations in sorting may equal, or even exceed the number of comparisons, e.g. the number of times an item is moved in memory, or the number of subscript comparisons. Any satisfactory analysis will have to include these factors as well.

An even more important deficiency is found in all currently available analytical studies, and that is that the analysis is carried out only for the simplified case of the unique, randomly ordered items. Thus these studies give but little help in the practical selection of a good sorting algorithm.

4. Empirical Tests

4.1 Preliminaries

In this section we develop empirical tests for the study of sorting algorithms. The basic idea is of course to conduct timing trials on an actual computer.
The results obtained in this way will naturally depend on the particular computer used for the tests. However, if we conduct the timing trials for two or more algorithms we can find out the relative advantage of one over the other, and, moreover, a similar relative advantage will generally hold for other computers as well.

4.2 Existing Order

The amount of existing order in the input data can be measured by the correlation with an ordered sequence of data. For data which are already in ascending order the correlation coefficient is 1, for data which are in reverse order the correlation coefficient is -1, and for data which are randomly ordered the correlation coefficient is 0.

Fortunately there is a simple method for generating data having any desired correlation with ordered data. If for $1 \leq i \leq N$, $A_i$ and $B_i$ are independent, normally distributed random sequences with mean 0 and standard deviation 1, and if

$$X_i = \nu A_i + \sqrt{1 - \nu^2} B_i,$$

then $A_i$ and $X_i$ are dependent, normally distributed sequences with mean 0, standard deviation 1, and correlation coefficient $\nu$. Thus if we sort the $X_i$ carrying along the $A_i$, we have the desired result. Specifically $A_i$ has correlation $\nu$ with an ordered sequence of data and is used as test data for the timing tests.

The best reference for methods of generating random numbers is Knuth [10], and formula (3) appears there in a more general form. A procedure similar to the above has also been used by Chambers [1] in the study of an algorithm for partial sorting.

4.3 Equal Items

The method for generating data containing many equal items is even simpler. To obtain a sample of data consisting of only $k$ distinct elements we generate a sequence of random integers uniformly distributed between 0 and $k-1$, and this se-
quence constitutes the test data. The technique for generating random integers is well known; see for example Knuth [10].

4.4 Experimental Results

The tests outlined above were performed for the Quicksort and Shellsort algorithms defined in section 2. The programs were run on a Control Data 6600 computer under the control of the UCMS 1.0 operating system. The times reported are averages calculated for a varying number of trials ranging from 4 to 4000. The times are subject to a 1% error due to the relatively low resolution of the time keeping apparatus in the operating system (1 millisecond).

The main results are reported in Tables I and II. We note that both Quicksort and Shellsort perform relatively satisfactorily with respect to all three timing criteria. We note in Table III however, that the running time of Shellsort grows slightly faster than $N \log_2 N$.

No auxiliary memory space is required by Shellsort beyond what can be stored in the fast registers of the computer. For Quicksort the amount of auxiliary memory space is bounded by $2 \log_2 N$. In a machine language program information can be packed so that the bound is actually $\log_2 N$. An interesting proof of this bound is given by Hibbard [5], p. 207. The largest memory currently available for the Control Data 6000/7000 series is $2^{17}$ cells, so that no more than 16 cells of auxiliary storage is required by Quicksort.

5. Conclusion

When sorting data composed of unique, randomly ordered items, the close agreement of the Quicksort method with the theoretical minimum number of comparisons has been noted by Hoare [7] and others. However, earlier versions of the method were
relatively inefficient with respect to the other criteria developed in this paper. On this account Quicksort was poorly suited as a general purpose sorting procedure. It has been shown that the Quicksort algorithm developed by Singleton has corrected these deficiencies and passes all of the tests. Furthermore Quicksort is faster than Shellsort in every instance and thus can be highly recommended. It is the logical standard of comparison for any sorting algorithm proposed in the future.

Acknowledgement

The author gratefully acknowledges the stimulation and sound technical advice received from Richard L. Hotchkiss and Philip A. Houle, both of the University Computer Center.
TABLE I

Average sorting times in milliseconds for sorting \( N \) items with correlation \( \rho \) with ordered data.

<table>
<thead>
<tr>
<th>( N )</th>
<th>( \rho )</th>
<th>-1.0</th>
<th>-0.5</th>
<th>0.0</th>
<th>+0.5</th>
<th>+1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Quicksort</td>
<td>1.6</td>
<td>2.2</td>
<td>2.2</td>
<td>2.3</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td>Shellsort</td>
<td>2.6</td>
<td>3.3</td>
<td>3.3</td>
<td>3.1</td>
<td>2.1</td>
</tr>
<tr>
<td>500</td>
<td>Quicksort</td>
<td>9.0</td>
<td>15</td>
<td>14</td>
<td>14</td>
<td>7.6</td>
</tr>
<tr>
<td></td>
<td>Shellsort</td>
<td>19</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>15</td>
</tr>
<tr>
<td>1000</td>
<td>Quicksort</td>
<td>19</td>
<td>32</td>
<td>31</td>
<td>29</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Shellsort</td>
<td>44</td>
<td>60</td>
<td>59</td>
<td>56</td>
<td>33</td>
</tr>
<tr>
<td>5000</td>
<td>Quicksort</td>
<td>100</td>
<td>190</td>
<td>190</td>
<td>190</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Shellsort</td>
<td>280</td>
<td>420</td>
<td>420</td>
<td>410</td>
<td>210</td>
</tr>
<tr>
<td>10000</td>
<td>Quicksort</td>
<td>250</td>
<td>390</td>
<td>400</td>
<td>440</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>Shellsort</td>
<td>610</td>
<td>970</td>
<td>980</td>
<td>970</td>
<td>460</td>
</tr>
</tbody>
</table>
TABLE II

Average sorting times in milliseconds for sorting \( N \) items chosen randomly from a set of \( k \) distinct elements.

<table>
<thead>
<tr>
<th>( N )</th>
<th>( k )</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td>2.0</td>
<td>1.8</td>
<td>1.8</td>
<td>1.9</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>Quicksort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shellsort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td></td>
<td>13</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Quicksort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shellsort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>Quicksort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shellsort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td></td>
<td>180</td>
<td>180</td>
<td>180</td>
<td>170</td>
<td>170</td>
</tr>
<tr>
<td></td>
<td>Quicksort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shellsort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td>390</td>
<td>390</td>
<td>380</td>
<td>380</td>
<td>380</td>
</tr>
<tr>
<td></td>
<td>Quicksort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shellsort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TABLE III

Average sorting times in microseconds divided by $N \log_2 N$, when sorting $N$ unique, randomly ordered items.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Quicksort</th>
<th>Shellsort</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>3.3</td>
<td>4.9</td>
</tr>
<tr>
<td>500</td>
<td>3.2</td>
<td>5.6</td>
</tr>
<tr>
<td>1000</td>
<td>3.1</td>
<td>5.9</td>
</tr>
<tr>
<td>5000</td>
<td>3.0</td>
<td>6.9</td>
</tr>
<tr>
<td>10000</td>
<td>3.0</td>
<td>7.4</td>
</tr>
</tbody>
</table>
REFERENCES


APPENDIX A

COMPASS (CDC 6000/7000 series)

Definition of Quicksort*

*A Compass program written for the CDC 6400 computer and distributed in the report cited below was useful to the author in developing the program which follows.

IDENT QSORT (A,N)
ENTRY QSORT
*
QUICKSORT.
* Sorts array A into ascending order, from A(1) to A(N).
* Assumes standard FORTRAN calling sequence.
* Ordering is by integer subtraction, thus overflow is possible when sorting large positive and negative numbers.
*
REGISTER ASSIGNMENTS:
* (H1) = LOC(A(I))
* (H2) = LOC(A(J))
* (H3) = LOC(A(K))
* (H4) = LOC(A(L))
* (H5) = M
* (H6) = LOC(A(1))
* (H7) = 1
*
PROGRAM BY STEVE LEGENHAUSEN, SEPTEMBER 1971.

I J
HSS 16 RECURSION ADDRESS ARRAY
QSORT EN **+200000B ENTRY
SA2 B2 N
MX0 43
SA4 1
X1 X2*X0
SH6 31
NZ X1*QSORT IF N < 0 Y N > 131071 THEN RET
SH2 31-37 M = 0
SH5 30 J = LOC(A(N))
SH2 X2*X2 J = LOC(A(N))

** ESTIMATE MEDIAN
L05 GE A1,A2,A70 IF I ≥ J THEN GOTO L70
L10 SX0 42=41 J = I
S63 31 K = I
SA4 1 (J - I) / 2
SN4 A2 X2+A2 L = J
SA4 B1 A(I)
SA4 A2 A(J)
SA3 B2 A(I)
SA5 X2 T = A(I)
X0 X5-X4 PL X0+L20 IF A(I) ≤ T THEN GOTO L20
X2 X4 A(IJ) = A(I)
X4 X5 A(I) = T
X5 X2 T = A(IJ)
X0 X3-X5 PL X0+L25 IF A(J) ≥ T THEN GOTO L25
X2 X3 A(IJ) = A(J)
X3 X5 A(J) = T
X5 X2 T = A(IJ)
X4 X2 A(IJ) = A(I)
PL X0+L25 IF A(I) ≤ T THEN GOTO L25
X4 X4 A(IJ) = A(I)
X4 X5 A(I) = T
X5 X2 T = A(IJ)
X6 X2
SA6 A2 A(IJ)
SA6 X3 A(L) = A(K)
SA7 D3 A(K) = T
SA2 B4=87
S40 d3+H7  A(L-1)  L = L - 1
S44 d4+H7  
I40 x5-x4
S42 d4+H7  IF A(L) > T THEN GOTO L40
L50 x0+L40  
I43 x1  
I40 x3-x5
S43 d3+H7  A(K+1)
S41 d3+H7  K = K + 1

IF A(K) < T THEN GOTO L50
L4 = d3+H4+L30  IF K ≤ L THEN GOTO L30

CONTROL

S44 b4+H1  L = I
S43 b2+H3  J = K
L5 d4+H3+L60  IF L - I ≤ J - K THEN GOTO L40
S40 b1  I
S47 d2+H7  L
L40 18
S41 A1+H7  I = K
H4 x0+x7
S46 b5+IJ  IJ(M) = (I+L)
S45 35+H7  M = M + 1
E4  L80  GOTO L80

L60  S40 A1+H7  K
S47  d2  J
L40  L80  GOTO L80

S40 A1+H7  K
S47  d2  J
L40  L80  GOTO L80

IF M < O THEN RETURN

S42 x1  J
A41 18
S41  x1  I
L60  S44 11
S43  d2+H1  IF J - I ≥ 11 THEN GOTO L10
G6  b3+H4+L10  IF I = II THEN GOTO L05
E4  L1+B6+L05

INTERCHANGE SORT

L90  S41  d1  A(I)
S45  b1+H7  T = A(I+1)
M  d1+H2+L70  IF I ≥ J THEN GOTO L70
I40  x5-x1
S41  b1+H7  I = I + 1
P4  x0+L90  IF A(I) ≤ T THEN GOTO L90
S43  b1+H7  K = I - 1
L100  M46  x1
S41  b3+H7  A(K-1)
S46  b3+H7  A(K+1) = 4(K)
I40  x5-x1
S43  b3+H7  K = K - 1
M46  x5
N4  x0+L100  IF A(K) > T THEN GOTO L100
S46  b3+H7  A(K+1) = T
E4  L90  GOTO L90
END
IDENT SSORT (A,N)
ENTRY SSORT

* SHELLSORT.
* SORTS ARRAY A INTO ASCENDING ORDER FROM A(I) TO A(N).
* ASSUMES STANDARD FORTRAN CALLING SEQUENCE.
* ORDERING IS BY INTEGER SUBTRACTION, THUS OVERFLOW IS POSSIBLE WHEN SORTING LARGE POSITIVE AND NEGATIVE NUMBERS.
* REGISTER ASSIGNMENTS:
* (11) = LOC(A(I))
* (12) = LOC(A(J))
* (13) = LOC(A(K))
* (14) = LOC(A(I))
* (15) = 4
* (16) = LOC(A(N))
* (17) = 1

PROGRAM BY STEVE LEGENHAUSEN, SEPTEMBER 1971.

SSORT E:1000008 ENTRY
S42 d2 N
M40 43
S47 1
S41 x2=x0
S46 x2
S4 N1 A1,SSORT IF N < 0 OR N > 131071 THEN RET
S44 d1 LOC(A(I))
S4 IF 2 + P < N <= 2 + (P + 1) THEN M = 2 + P - 1.
S8 x5 b7+b7 = M = 2 * M
L10 S8 x5 b5-b5
S86 b6-b4
S6 x5 b5-b5
S4 IF M < N THEN GOTO L10
S4 NEXT PASS
L20 S5 x0 d5 = M = M / 2
S4x0 1
S4 x5 0
S4 z4 x0,SSORT IF M = 0 THEN RETURN
S4 z2 d4
S4 z3 b6-b5 = J = 1
S4 z3 b6-b5 = K = N - M
S4 PRIMARY LOOP
L30 S4 a1 a2 A(J)
S4 a5 b2+b5 A(J+M)
G7 b2+b3+b20 IF J > K THEN GOTO L20
I2 x0 x5=x1
S4 x2 b2+b7 = J = J + 1
PL x0 x3+b7 IF A(J) < A(J+M) THEN GOTO L30
S4 x1 x5=x1
S4 x6 x1
S45 x0 x3+b7 IF A(J) > T THEN GOTO L50
S4 x6 x5
S46 b1+b5 A(I+M) = A(I)
S4 x6 x5
S46 b1+b5 A(I+M) = A(I)
S4 x6 x5
S46 b1+b5 A(I+M) = T
EW L30 GOTO L30
END