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

ABSTRACT

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 auxilliary 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

$$\log_2 N! \tag{1}$$

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

$$N! \approx \sqrt{2\pi N}\,(N/e)^N.$$

With a good slide rule we readily calculate

$$\log_2 N! \approx N\log_2 N - 1.44N + .50\log_2 N + 1.33 \tag{2}$$

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 auxilliary 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 auxilliary 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

$$X_i = \rho A_i + \sqrt{1 - \rho^2}\, B_i , \tag{3}$$

then A_i and X_i are dependent, normally distributed sequences with mean 0, standard deviation 1, and correlation coefficient ρ. Thus if we sort the X_i carrying along the A_i, we have the desired result. Specifically A_i has correlation ρ 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 NCMS 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 apartus 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 auxilliary memory space is required by Shellsort beyond what can be stored in the fast registers of the computer. For Quicksort the amount of auxilliary 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 auxilliary 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 ρ with ordered data.

N	Sort \ ρ	-1.0	-0.5	0.0	+0.5	+1.0
100						
	Quicksort	1.6	2.2	2.2	2.3	1.3
	Shellsort	2.6	3.3	3.3	3.1	2.1
500						
	Quicksort	9.0	15	14	14	7.6
	Shellsort	19	25	25	25	15
1000						
	Quicksort	19	32	31	29	18
	Shellsort	44	60	59	56	33
5000						
	Quicksort	100	190	190	190	100
	Shellsort	280	420	420	410	210
10000						
	Quicksort	250	390	400	440	200
	Shellsort	610	970	980	970	460

TABLE II

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

N	Sort \ k	1	2	4	8	16
100						
	Quicksort	2.0	1.8	1.8	1.9	2.0
	Shellsort	2.1	2.2	2.4	2.6	2.8
500						
	Quicksort	13	12	12	12	12
	Shellsort	15	16	17	18	20
1000						
	Quicksort	29	28	27	27	28
	Shellsort	33	36	38	41	44
5000						
	Quicksort	180	180	180	170	170
	Shellsort	220	230	240	260	270
10000						
	Quicksort	390	390	380	380	380
	Shellsort	470	500	520	560	590

TABLE III

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

N	Quicksort	Shellsort
100	3.3	4.9
500	3.2	5.6
1000	3.1	5.9
5000	3.0	6.9
10000	3.0	7.4

REFERENCES

1. Chambers, J. M. Algorithm 410, Partial sorting. Comm. ACM 14, 5 (May 1971), 357-358.

2. Frank, R. M. and Lazarus, R. B. A high-speed sorting procedure. Comm. ACM 3, 1 (January 1960), 20 - 22.

3. Frazer, W. D. and McKellar, A. C. Samplesort: A sampling approach to minimal storage tree sorting. J. ACM 17, 3 (July 1970), 496 - 507.

4. Hibbard, Thomas N. Some combinatorial properties of certain trees with applications to searching and sorting. J. ACM 9, 1 (January 1962), 13 - 28.

5. Hibbard, Thomas N. An empirical study of minimal storage sorting. Comm. ACM 6, 5 (May 1963), 206 - 213.

6. Hoare, C. A. R. Algorithm 63, Partition, and Algorithm 64, Quicksort. Comm. ACM 4, 7 (July 1961), 321.

7. Hoare, C. A. R. Quicksort. Computer J. 5, 1(April 1962), 10 - 15.

8. Hurwitz, H. Jr. On the probability distribution of the values of binary trees. Comm. ACM 14, 2 (February 1971), 99 - 102.

9. Isaac, E. J. and Singleton, R. C. Sorting by address calculation. J. ACM 3, 3 (July 1956), 169 - 174.

10. Knuth, Donald E. The Art of Computer Programming, Volume 2, Seminumerical Algorithms. Addison - Wesley, Reading, Mass., 1969.

11. Morris, Robert. Some theorems on sorting. SIAM J. Appl. Math. 17, 1 (January 1969), 1 - 6.

12. Scowen, R. S. Algorithm 271, Quickersort. Comm. ACM 8, 11 (November 1965), 669 - 670.

13. Shell, D. L. A high - speed sorting procedure. Comm. ACM 2, 7 (July 1959), 30 - 32.

14. Singleton, Richard C. Algorithm 347, An efficient algorithm for sorting with minimal storage. Comm. ACM 12, 3 (March 1969), 185 - 187.

15. van Emden, M. H. Increasing the efficiency of Quicksort. Comm. ACM 13, 9 (September 1970), 563 - 567.

16. van Emden, M. H. Algorithm 402, Increasing the efficiency of Quicksort. Comm. ACM 13, 11 (November 1970), 693 - 694.

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.

Singleton, Richard C. An efficient algorithm for sorting with minimal storage. Research Memorandum, Stanford Research Institute, Mathematics and Statistics Division (September 1968), 16 pp.

```
          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 POS-
*   SIBLE WHEN SORTING LARGE POSITIVE AND NEGATIVE NUMBERS.
* REGISTER ASSIGNMENTS:
*   (B1) = LOC(A(I))
*   (B2) = LOC(A(J))
*   (B3) = LOC(A(K))
*   (B4) = LOC(A(L))
*   (B5) = M
*   (B6) = LOC(A(1))
*   (B7) = 1
* PROGRAM BY STEVE LEGENHAUSEN, SEPTEMBER 1971.
 IJ       BSS    16           RECURSION ADDRESS ARRAY
 QSORT    EQ     *+200000B    ENTRY
          SA2    B2           N
          MX0    43
          SB7    1
          BX1    X2*X0
          SB6    B1           II = I
          NZ     X1,QSORT     IF N < 0 V N > 131071 THEN RET
          SB2    B1-B7
          SB5    B0           M = 0
          SB2    X2+B2        J = LOC(A(N))
* ESTIMATE MEDIAN
 L05      GE     B1,B2,L70    IF I ≥ J THEN GOTO L70
 L10      SX0    B2-B1        J - I
          SB3    B1           K = I
          AX0    1            (J - I) / 2
          SB4    B2           L = J
          SA2    X0+B1        A(IJ)
          SA4    B1           A(I)
          SA3    B2           A(J)
          BX5    X2           T = A(IJ)
          IX0    X5-X4
          PL     X0,L20       IF A(I) ≤ T THEN GOTO L20
          BX2    X4           A(IJ) = A(I)
          LX4    X5           A(I) = T
          BX5    X2           T = A(IJ)
 L20      IX0    X3-X5
          PL     X0,L25       IF A(J) ≥ T THEN GOTO L25
          IX0    X3-X4
          BX2    X3           A(IJ) = A(J)
          LX3    X5           A(J) = T
          BX5    X2           T = A(IJ)
          PL     X0,L25       IF A(I) ≤ T THEN GOTO L25
          BX2    X4           A(IJ) = A(I)
          LX4    X5           A(I) = T
          BX5    X2           T = A(IJ)
 L25      LX6    X2
          SA6    A2           A(IJ)
* SPLIT
 L30      BX6    X3
          LX7    X4
          SA6    B4           A(L) = A(K)
          SA7    B3           A(K) = TT
          SA2    B4-B7
```

```
                SA1     B3+B7
L40             BX4     X2              A(L-1)
                SB4     B4-B7           L = L - 1
                IX0     X5-X4
                SA2     B4-B7
                NG      X0,L40          IF A(L) > T THEN GOTO L40
L50             BX3     X1              A(K+1)
                SB3     B3+B7           K = K + 1
                IX0     X3-X5
                SA1     B3+B7
                NG      X0,L50          IF A(K) < T THEN GOTO L50
                LE      B3,B4,L30       IF K ≤ L THEN GOTO L30
* CONTROL
                SB4     B4-B1           L - I
                SB3     B2-B3           J - K
                LE      B4,B3,L60       IF L - I ≤ J - K THEN GOTO L60
                SX0     B1              I
                SX7     A2+B7           L
                LX0     18
                SB1     A1-B7           I = K
                BX6     X0+X7
                SA6     B5+IJ           IJ(M) = (I,L)
                SB5     B5+B7           M = M + 1
                EQ      L80             GOTO L80
L60             SX0     A1-B7           K
                SX7     B2              J
                LX0     18
                SB2     A2+B7           J = L
                BX6     X0+X7
                SA6     B5+IJ           IJ(M) = (K,J)
                SB5     B5+B7           M = M + 1
                EQ      L80             GOTO L80
L70             SB5     B5-B7           M = M - 1
                SA1     B5+IJ           IJ(M)
                LT      B5,B0,QSORT     IF M < 0 THEN RETURN
                SB2     X1              J
                AX1     18
                SB1     X1              I
L80             SB4     11
                SB3     B2-B1
                GE      B3,B4,L10       IF J - I ≥ 11 THEN GOTO L10
                EQ      B1,B6,L05       IF I = II THEN GOTO L05
* INTERCHANGE SORT
L90             SA1     B1              A(I)
                SA5     B1+B7           T = A(I+1)
                GE      B1,B2,L70       IF I ≥ J THEN GOTO L70
                IX0     X5-X1
                SB1     B1+B7           I = I + 1
                PL      X0,L90          IF A(I) ≤ T THEN GOTO L90
                SB3     B1-B7           K = I - 1
L100            BX6     X1
                SA1     B3-B7           A(K-1)
                SA6     B3+B7           A(K+1) = A(K)
                IX0     X5-X1
                SB3     B3-B7           K = K - 1
                BX6     X5
                NG      X0,L100         IF A(K) > T THEN GOTO L100
                SA6     B3+B7           A(K+1) = T
                EQ      L90             GOTO L90
                END
```

```
          IDENT  SSORT (A,N)
          ENTRY  SSORT
* SHELLSORT.
* 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 POS-
*   SIBLE WHEN SORTING LARGE POSITIVE AND NEGATIVE NUMBERS.
* REGISTER ASSIGNMENTS:
*   (B1) = LOC(A(I))
*   (B2) = LOC(A(J))
*   (B3) = LOC(A(K))
*   (B4) = LOC(A(1))
*   (B5) = M
*   (B6) = LOC(A(N))
*   (B7) = 1
* PROGRAM BY STEVE LEGENHAUSEN, SEPTEMBER 1971.
SSORT     EQ     *+200000B     ENTRY
          SA2    B2            N
          MX0    43
          SB7    1
          BX1    X2*X0
          SB6    X2
          NZ     X1,SSORT      IF N < 0 V N > 131071 THEN RET
          SB4    B1            LOC(A(1))
* IF 2 ↑ P < N ≤ 2 ↑ (P + 1) THEN M = 2 ↑ P - 1.
          SB5    B7+B7         M = 2
L10       SB5    B5+B5         M = 2 * M
          LT     B5,B6,L10     IF M < N THEN GOTO L10
          SB6    B6+B4         N = LOC(A(N))
          SB5    B5-B7         M = M - 1
          SB6    B6-B7
* NEXT PASS.
L20       SX0    B5            M = M / 2
          AX0    1
          SB5    X0
          ZR     X0,SSORT      IF M = 0 THEN RETURN
          SB2    B4            J = 1
          SB3    B6-B5         K = N - M
* PRIMARY LOOP.
L30       SA1    B2            A(J)
          SA5    B2+B5         A(J+M)
          GT     B2,B3,L20     IF J > K THEN GOTO L20
          IX0    X5-X1
          SB2    B2+B7         J = J + 1
          PL     X0,L30        IF A(J) ≤ A(J+M) THEN GOTO L30
          SB1    B2-B7         I = J - 1
          BX6    X1
* SECONDARY LOOP.
L40       SA6    B1+B5         A(I+M) = A(I)
          SB1    B1-B5         I = I - M
          LT     B1,B4,L50     IF I < 1 THEN GOTO L50
          SA1    B1            A(I)
          IX0    X5-X1
          BX6    X1
          NG     X0,L40        IF A(I) > T THEN GOTO L40
L50       BX6    X5
          SA6    B1+B5         A(I+M) = T
          EQ     L30           GOTO L30
          END
```