Proceedings of the 1987 Fifteenth Annual ACM Computer Science Conference, St. Louis, Mo., Feb. 19-20, 1987, pp. 183-190.

QUICKSORT ALGORITHMS WITH AN EARLY EXIT FOR SORTED SUBFILES

Roger L. Wainwright
The University of Tulsa

ABSTRACT

The Quicksort algorithm is known to be one of the most efficient internal sorting techniques. Quicksort has received considerable attention almost from the moment of its invention. This paper reviews some of the important improvements to Quicksort that have appeared in the literature. Historically, the improvements to Quicksort have been in one of the following areas: (1) algorithms for determining a better pivot value, (2) algorithms that consider the size of the generated subfiles, and (3) various schemes used to partition the file. Despite improvements in these areas, the worst case situation of $O(n^2)$ comparisons for sorted or nearly sorted files still remains. This paper proposes a fourth research area for Quicksort improvement designed to remove the worst case behavior due to sorted or nearly sorted files. During the partitioning process (using any scheme) determine if the left and right subfiles are in sorted order. This is a minor but very effective modification to the Quicksort algorithm. A new Quicksort algorithm, Qsorte, is presented that provides an early exit for sorted subfiles. Test results on randomly generated lists, nearly sorted lists, sorted and sorted lists in reverse order are given for Quicksort, Quickersort, Bsort, Qsorte and several other algorithms. Results show Qsorte performs just as well as Quicksort for random files and in addition has O(n) comparisons for sorted or nearly sorted files and O(n) comparisons for sorted or nearly sorted files in reverse.

INTRODUCTION

The Quicksort algorithm introduced by C.A.R. Hoare in 1961 [6] is probably the most widely used sorting algorithm. It is widely accepted as the most efficient internal sorting technique. There are several desirable features of the Quicksort algorithm. First, it sorts in-place using only a small auxiliary stack to keep track of the unsorted subfiles. Secondly, it requires $O(n \log_2 n)$ operations to sort n items and finally, it is fairly easy to implement.

Quicksort sorts a list of keys A[1], A[2],..., A[n] recursively by choosing a key v in the list as a pivot key around which to rearrange the other keys in the list. Ideally, the pivot key is near the median key value in the list, so that it is preceded by about half of the keys and followed by the other half. The keys of the list are now rearranged such that for some k, A[1], A[2],..., A[k] contain all the keys with values less than v, and A[k + 1], A[k + 2],..., A[n] contain all the keys with values greater than or equal to v. The elements A[1], A[2],...,A[k] are called the left sublist and the elements A[k + 1], A[k + 2],..., A[n] are the right sublist. Thus, the original list is partitioned into two sublists where all the keys of the left sublist precede all the keys of the right sublist. After partitioning, the original problem of sorting the entire list is now reduced to the problem of sorting the left and right sublists independently. The Quicksort algorithm based on a recursive approach is given below:

```
Procedure Quicksort (m,n:integer);
{sort elements A[m],..., A[n] of global array A}
VAR k:integer; {location of the partition}
    v:integer; {pivot value}
BEGIN
if m < n then
    FINDPIVOT (m,n,v);
    PARTITION (m,n,k);
    QUICKSORT (m,k-1);
    QUICKSORT (k,n);
END; {QUICKSORT}
```

The main drawback of Quicksort is the worst case behavior of $O(n^2)$ comparisons. Suppose the values in A[1], A[2],..., A[N] are in sorted order and A[1] is used as the pivot value. A file of size n will be partitioned into subfiles of sizes 1 and N-1. The subfile of size n-1 will be partitioned into subfile sizes of 1 and N-2, and so on. This same problem also arises when the file is sorted in resverse order. It has been shown in [17] that Quicksort also performs poorly for nearly sorted files or nearly sorted in reverse. This is a very disappointing aspect of Quicksort since files with these distributions are very

likely to occur in practice.

Quicksort has received considerable attention almost from the moment of its invention. There have been numerous improvements for Quicksort appearing in the literature. Several improvements have been suggested for the pivot value. Hoare's Quicksort uses a randomly chosen element from the file as the pivot value [6,7]. In 1965, Scowen's Quickersort algorithm improved on Quicksort by selecting the middle element of the array as the pivot value [11]. He points out the best possible value for the pivot would be one which splits the file into two halves of equal size. Thus, if the file is sorted or nearly sorted the middle element will prove to be an excellent choice. If the file is completely random, then the middle element is as good as any other. Thorough investigation has verified that choosing the middle element is generally better than cloosing a random element as the pivot. Loeser [9] gives some performance evaluations for Quicksort, Quickersort and several other sorting algorithms. In 1969, Singleton suggested a median-of-three method to obtain a better pivot value [14]. We choose the left, middle and right elements from the file. Sort these elements and place the values back into the same positions in the file. The median of the three values becomes the pivot. This technique guarantees that after splitting a file of size n the longest subfile will be less than or equal to n-2, rather than n-1 as in Quicksort or Quickersort. The median-of-three method helps Quicksort in three ways. First, it makes the worst case situation more unlikely to occur. Second, if the middle element (pivot) is interchanged with the value in the rightmost-1 position then the need for a sentinel value for partitioning is eliminated. Third, and most importantly, it reduces the total running time of the algorithm by about 5% [13]. It is possible to extend this technique to more than three elements; for example a median-of-five partitioning. Another technique for determining a pivot value is reported by Aho and others [1]. The subfile is scanned left to right looking for two different values. The larger of the leftmost two different keys becomes the pivot. The advantage of this technique is that sentinel values are not needed. Another advantage is that if two different key values cannot be found then the file contains all equal values and therefore is sorted. In Meansort [10] Motzkin calculates the mean value for each subfile and uses that as the pivot. This is one of the few variations of Quicksort that does not use one of the key values in the file as the pivot. For most distributions, the value of the mean will be a better estimate of the median than any key selected at random. Motzkin observed that the sorting time of Meansort is independent of the order of the keys. The sorting time depends only on the distribution of their values. Thus, files already in sorted order or sorted in reverse order will be sorted in the same time as if the key values were shuffled in any arbitrary way. This virtually eliminates the worst case situation of $O(n^2)$ operations except for some rare distributions of keys. However, further study of Meansort in [17] showed it to be very inefficient. The additional cost in calculating the mean values for each subfile seem to outweight the advantage.

Several improvements to Quicksort have been suggested by considering the size of the subfiles. In 1962, Hillmore [5] pointed out it is better not to split subfiles of size two or less, just sort them instead. Both Hibbard [4] and Scowen [11] point out that the length of the auxiliary stack is minimal if the smaller subfile is split and the index position of the larger subfile is placed in the stack. That is, we should process the smaller subfile first. Several investigators have noted that Quicksort is not very efficient for small subfiles. This is most unfortunate since the recursive nature of Quicksort quarantees that many small subfiles will be generated. Hoare [7] was the first to suggest that small subfiles be sorted by some other method. He suggested insertion sort. Knuth [8] suggests nine as the size subfile on which Quicksort should invoke a simpler sorting algorithm. This will also reduce the size of the auxiliary stack. The size of the subfile depends on several factors. These include the time spent making a recursive call, machine architecture, the implementation strategy for procedure calls used by the compiler of the language the sort is written in, as well as other factors [1]. Sedgewick [12] suggests another strategy for small subfiles. Subfiles of size $\leq M$ are ignored and not partitioned any further. When the program finishes, the file will not be sorted. Instead the file will be nearly sorted in that it will contain small groups of randomly ordered values, such that the elements in any one group will all be less than elements in any group to the right. We simply sort the entire array by insertion sort. Sedgewick found the best value for M was about 9 or 10. Since M is implementation dependent, he suggests any value between 6 and 15 would do about as well. Bently [2] found for his system the best value for M was 15. Sedgewick further observed that on the average the auxiliary stack size was reduced by 75%. This makes this technique preferable to the scheme of sorting smaller subfiles first in order to reduce the auxiliary stack size.

There have been many partitioning methods suggested for Quicksort. In Qsort [16] Van Emden uses two pivot values during the partitioning process. This results in left and right subfile and two keys in a middle subfile rather than just one middle key for most other implementations. Standish [15] uses a partitioning method where the pivot value is involved in all of the interchanges. The most commonly used partitioning scheme, however, is due to Sedgewick and is based on two approaching indices. Bently [2] presents a scheme due to Lomuto where both indices begin at the left end of the file and move towards the right end. There seems to be no end to the variety of partitioning schemes found in the literature. Most are slight variations of each other with little difference in performance. Some, however, greatly improve the performance of Quicksort.

We have reviewed the development of Quicksort and some of its variations. Historically, we have seen that the improvements to Quicksort have occurred in one of the following areas: (1) algorithms for determining a "better" pivot value, (2) algorithms that consider the size of

the generated subfiles, and (3) various schemes used to partition the file. In this paper, we suggest a fourth area for improvement. During the partitioning process (using any scheme) determine if the left and right subfiles are in sorted order. To this end, we suggest the following minor but very effective modification to the basic Quicksort algorithm. Let lsorted and rsorted be boolean variables such that lsorted is true when the left subfile is in sorted order and false otherwise. Rsorted is true when the right subfile is in sorted order and false otherwise.

```
Procedure Quicksort (m,n:integer);
VAR k,v:integer;
    lsorted, rsorted: boolean;
BEGIN
If m < n then
 FINDPIVOT (m,n,v);
    PARTITION (m,n,k, lsorted, rsorted);
    If not lsorted then Quicksort (m,k-1);
    If not rsorted then Quicksort (k,n);
END; {Quicksort}
```

This idea was first suggested in [17] where three different algorithms were presented, Bsort, Xsort and Ysort. Xsort and Ysort are minor variations of Bsort. Each algorithm uses a slightly different technique to determine if a subfile is in sorted order. To preserve continuity, we review the Bsort algorithm before presenting a new algorithm, Qsorte.

THE ALGORITHM FOR BSORT (REVIEWED)

As in Quickersort, Bsort selects the middle key during each pass as the pivot key for controlling the partitioning process. The algorithm then proceeds as traditional Quicksort. Each key that is placed into the left subfile is done so at the right end of that subfile. If the new key is not the first key of the subfile, a comparison is made between the new key and its left neighbor to ensure that the pair of keys is in sorted order; if not, they are then interchanged. Similarly, each new key that is placed into the right subfile is done so at the left end of that subfile. If the new key is not the first key of the subfile, a comparison is made between the new key and its right neighbor to ensure the pair of keys is in sorted order; if not, they are interchanged. Thus, at any point in the construction of the left subfile, the rightmost key will be the largest in value, and at any point in the construction of the right subfile, the leftmost key will be the smallest in value.

When the partitioning process is finished and there were no interchanges performed during the construction of a subfile, then its keys are in sorted order and further partitioning is not needed. In addition, if a subfile has only two keys, then by the above method of construction it is sorted. If a subfile has three keys, it can be sorted by making one comparison and possibly one interchange. A subfile will not be partitioned further unless it has at least four keys. The algorithm is finished when all of the subfiles are sorted.

THE ALGORITHM FOR QSORTE

Qsorte (Quicksort with early exit for sorted subfiles) uses the middle key during each pass to control the partitioning process. Initially each subfile is marked as being sorted. When a new key is placed into the left subfile and the subfile is still marked as sorted, then if the new key is not the first key a comparison is made between the new key and its left neighbor. So far this is the same as Bsort. However, unlike Bsort if the keys are not in sorted order then the subfile is marked as unsorted. The keys are not interchanged. Similarly, when a new key is placed into the right subfile and the subfile is marked as sorted, then if the new key is not the first key (rightmost) in the subfile a comparison is made between the new key and its right neighbor. If the keys are not in sorted order, we mark the subfile as unsorted. When the partitioning process is finished, any subfile that is marked as sorted will not be partitioned further. The method of processing smaller subfiles first may be applied in order to minimize the stack. The algorithm is finished when all of the subfiles are sorted. (A Pascal version of the Qsorte algorithm is given in the Appendix.) Sentinel values may be used to avoid checking if a value is the first one in the subfile. In the Qsorte algorithm this means that the comparisons i > m and j < n may be omitted when sentinel values are used. We determined sentinel valids did not improve the performance of Qsorte on random files. The following example illustrates how the Qsorte algorithm works.

Consider the following file with the middle key, 40 as the pivot key controlling the partitioning process. We start with two pointers, i and j initially pointing one position left and one position to the right of the subfile, respectively. We use p as a pointer to the position of the pivot value. The significance of this pointer will be explained later.

```
                    p
   12 15 97 26 77 30 40 57 86 31 28 71 21
i                                          j
```

We move i to the right until a number larger to or equal to the pivot value is found. In this case, i will rest on 97. Note during this process 12 and 15 are compared and the left subfile is still in sorted order. Next we move j to the left until a number less than the pivot value is found. This yields the following:

```
                    p
   12 15 97 26 77 30 40 57 86 31 28 71 21
      i                                j
```

Exchange the keys pointed to by i and j.

```
                    p
   12 15 21 26 77 30 40 57 86 31 28 71 97
      i                                j
```

Now 15 and 21 are compared and the left subfile is still in sorted order. Next move i to the right again until it rests on a value larger than or equal to the pivot; the value in this case is 77. In the process, keys 21 and 26 are compared and

the left subfile remains in sorted order. Move j left until it rests on a value less than the pivot (in this case 28). In the process, keys 71 and 97 are compared and the right subfile is in sorted order. This yields

```
                    p
12 15 21 26 77 30 40 57 86 31 28 71 97
            i                 j
```

We exchange keys pointed to by i and j yielding

```
                    p
12 15 21 26 28 30 40 57 86 31 77 71 97
            i                 j
```

After the exchange keys 26 and 28 are compared. The left subfile remains in sorted order. Keys 77 and 71 are also compared and the right subfile is marked as not sorted; thus the right subfile will not be tested in further instances for sortedness. Next we move i right to 40 noting the left subfile is still sorted, and move j left to 31. We exchange keys 40 and 31 yielding

```
                          p
12 15 21 26 28 30 31 57 86 40 77 71 97
                  i        j
```

Note since the pivot value moved, p moves also. Keys 30 and 31 are in the proper order and the left subfile remains sorted. Keys 40 and 77 are not compared since we earlier determined the right subfile was not in sorted order. Finally, we move i right to 57 and j left to 31 yielding:

```
                          p
12 15 21 26 28 30 31 57 86 40 77 71 97
                  j  i
```

This method of partitioning the subfile always terminates with j one position to the left of i. Thus the left subfile ends in position j and the right subfile begins in position i as shown below

```
                          p
[12 15 21 26 28 30 31] [57 86 40 77 71 97]
                  j     i
```

As a final step, if the right subfile is not sorted we exchange keys in position p and i yielding the final result:

[12 15 21 26 28 30 31] 40 [86 57 77 71 97].

Since the left subfile is sorted, further partitioning of that subfile is not needed. The mechanism of keeping track of the pivot value and as a final step exchanging with the key in position i guarantees a file of size n will be divided into subfiles of no worse than sizes 1 and n-1. If the pivot pointer is not maintained, then it is possible to divide a file of size n into two subfiles of sizes zero and n (empty left subfile). Once this happens, an infinite loop is started. The following file illustrates this point. (This is the same file only the pivot value is 8 rather than 40.)

```
                    p
12 15 97 26 77 30 8 57 86 31 28 71 21
i                                   j
```

We move i right to 12 and j left to a position one left of the start of the file. The algorithm employs a check to ensure j does not move left beyond this position. This yields

```
                    p
12 15 97 26 77 30 8 57 86 31 28 71 21
j  i
```

Thus it becomes necessary to interchange the keys at positions p and i after the partitioning process is over. Most Quicksort algorithms remedy this problem by interchanging the pivot key with the leftmost key in the file before the partitioning begins. In our case, this approach does not work very well. If a file is sorted, nearly sorted, sorted in reverse or nearly sorted in reverse, moving the pivot key before the partitioning process begins will destroy the sortedness property of the file and increase the processing time. For a random distribution of keys within a file, this issue is not relevant; however, Qsorte is designed to detect and take advantage of sorted subfiles, while at the same time maintain the same excellent performance as Quicksort for random files. Qsorte does have a worse case situation, however, and is still of $O(n^2)$. For example, the following file was specifically generated such that in each instance the pivot value happens to be the smallest value in the subfile. In this case Qsorte repeatedly divides the subfiles into two subfiles where the size of one of the subfiles is always one. The reader is encouraged to try this.

2 8 4 12 6 10 1 3 5 7 9 11 13.

EMHANCEMENTS TO QSORTE

The enhancements to the Quicksort algorithm that have been developed over the years can also be applied to the basic Qsorte algorithm as well. In this section, we will describe two common Quicksort enhancements and how this may be applied to the Qsorte algorithm. One enhancement determinines a better pivot value and the other considers the size of the generated subfiles.

QSORTEM. Qsortem is a combination of Qsorte and the median-of-three technique for determining a better pivot value. Before the partitioning process begins, the left, middle and right elements of each file are sorted and the values are placed back into the same three positions in the file. The middle key is the pivot key and the process continues the same as Qsorte. Note the median-of-three processes does not destroy any sortedness property of the file, in fact, any such property will be enhanced.

QSORTEMI. Qsortemi is a combination of Qsortem and the use of insertion sort for small subfiles. In our system, we have determined for subfiles of size 10 or less it is better (less C.P.U. time) to employ insertion sort than to continue to partition the subfile.

EMPIRICAL TEST RESULTS

Test results for the following six algorithms - Quicksort, Quickersort, Bsort, Qsorte, Qsortem and Qsortemi - are compared and discussed below.

Table I gives experimental results for the six algorithms acting on data sets of 2000 elements. The table shows the number of interchanges and comparisons made on list items, the number of partition steps required for the sort, and CPU time in seconds.

The computer programs for each algorithm were written in Pascal, and each program was run on the same computer. A personal computer was used to ensure that nothing else but the sorting program would compete for CPU resources. The CPU times include only the processing times for each algorithm: they include neither the time consumed in I/O activity nor the additional CPU time needed to determine the number of interchanges, comparisons, and/or partition steps. CPU times are accurate to .01 seconds.

Data set 1 is an ordered list, and data set 2 is an ordered list in reverse order. Data set 3 is a specially constructed data set designed as a worst case for algorithms using the middle key as the pivot value. Ignoring the middle element, the data set is an ordered list. The middle element is the largest element in the data set. Data set 4 is composed of numbers of the same value. Data sets 5 and 6 consist of lists with a sortedness ratio of 2 per cent and 2 per cent in reverse order, respectively. Data sets 7 and 8 are lists with a sortedness ratio of 6 per cent and 6 per cent in reverse order, respectively. Data sets 9 and 10 are lists with a sortedness ratio of 10 per cent and 10 per cent in reverse order respectively, while data sets 11-13 are random sets.

We define the sortedness ratio of a list with N records as k/N, where k is the minimum number of elements that must be removed so that the remaining portion of the list is sorted. For a sorted list, this ratio is zero, and for a completely out-of-order list, the ratio approaches one. For example, list 1, 2, 4, 5, 6, 3, 7, 8, 9 has a sortedness ratio of 1/9 and list 9, 8, 7, 3, 6, 5, 4, 2, 1 has a sortedness ratio of 1/9 in reverse order. Each list of size N with sortedness ratio k/N was created from the identity permutation: First, k randomly chosen elements were removed from the permutation and randomly inserted into another list of size N. Then the remaining N - k elements of the permutation were inserted in order in the vacant list slots. If in so doing, one of the random k elements lies between, and has a value between, two of the inserted elements, then the random element and one of the inserted elements are exchanged. Thus, the k elements of the permutation are the smallest subset whose removal leaves the list sorted; hence, the sortedness ratio of the list is k/N. This is the same definition and implementation of sortedness ratio that is described in [3].

Comparing CPU times among Qsorte, Qsortem and Qsortemi shows Qsortemi to be slightly superior to Qsortem and Qsortem to be slightly superior to Qsorte over nearly all of the data sets tested. These results were expected since it is well known that the median-of-three technique and using insertion sort for smaller subfiles improves the standard quicksort algorithm. These techniques also improve Qsorte.

Results show Quickersort yields a slightly higher number of interchanges than Quicksort over all of the datasets tested. However, the number of comparisons is significantly higher for Quicksort compared to Quickersort for data sets 1-10 (sorted or nearly sorted files in ascending and descending order) while relatively equal for random files (data sets 11-13). As expected Quickersort yielded better C.P.U. times for data sets 1-10 but worse CPU times for random files. On an overall basis, results show Quickersort to be a better algorithm than Quicksort. Both algorithms, however, have a worst case situation of $O(n^2)$ comparison as shown in data sets 1-3.

Qsorte and Bsort algorithms both employ an early exit check for sorted subfiles. Bsort requires more interchanges and comparisons than Qsorte. This is because of the "bubble type" interchange that Bsort performs during the partitioning process. As a result, Bsort requires less partition steps than Qsorte. However, comparing CPU times between Bsort and Qsorte clearly shows Qsorte as the superior algorithm.

Results show Qsorte performed less interchanges and comparisons, required less partitions and yielded significately better C.P.U. results compared to Quicksort over data sets 1-10. For random data sets, the number of interchanges, comparison and partition steps were relatively equal for Qsorte and Quicksort. Quicksort yielded only slightly superior C.P.U. times compared to Qsorte for random files. On an overall basis, test results show Qsorte to be a better algorithm than Quicksort.

SUMMARY AND CONCLUSIONS

We have reviewed several of the important improvements to Quicksort including algorithms for determining a better pivot value, algorithms that consider the size of the subfiles generated and various methods used to partition the file. We have proposed a new area of research for improvements to Quicksort: algorithms that detect sorted subfiles. We presented a new Quicksort algorithm, Qsorte, that provides an early exit for sorted subfiles. Our test results showed that Qsorte performed just as well as Quicksort and Quickersort for random files, with the added benefit of only O(n) comparisons for sorted or sorted in reverse files. Qsorte also performed better than Quicksort and Quickersort for nearly sorted files and nearly sorted in reverse. Our test results showed the performance of Quicksort was extremely poor for nearly sorted lists or nearly sorted in reverse. Quickersort also exhibited $O(n^2)$ comparisons as well under certain circumstances. In general, results show that the Qsorte algorithm is a considerable improvement over Quicksort. Qsorte had a much better average behavior over all of the data sets tested than both Quicksort and Quickersort. Furthermore,

TABLE I. Comparison of Sorting Algorithms on 2000 Elements

INTERCHANGES

DATA SETS	QUICKSORT	QUICKERSORT	BSORT	QSORTE	QSORTEM	QSORTEMI
1	1,999	1,998	0	0	0	0
2	1,999	3,706	1,000	1,000	1,002	1,002
3	1,999	3,997	1,001	1,001	1,128	1,106
4	9,893	10,915	0	0	0	0
5	2,498	3,363	8,257	2,067	2,541	3,002
6	3,171	4,032	6,681	2,573	2,836	2,966
7	3,043	4,055	12,021	2,587	3,433	4,335
8	4,151	4,574	13,234	13,278	3,858	4,565
9	3,313	4,341	13,976	2,960	3,775	4,883
10	3,932	4,757	14,347	3,466	4,075	4,936
11	5,221	6,342	23,758	5,129	5,746	6,836
12	5,157	6,269	29,824	5,001	5,746	6,806
13	5,146	6,356	23,339	5,105	5,820	6,909

COMPARISONS

DATA SETS	QUICKSORT	QUICKERSORT	BSORT	QSORTE	QSORTEM	QSORTEMI
1	2,002,998	20,010	4,000	4,000	4,001	4,001
2	2,002,998	21,474	6,000	4,000	4,001	4,001
3	2,002,998	2,002,998	8,000	24,684	26,186	24,836
4	19,786	19,786	4,001	4,001	4,003	4,003
5	244,851	22,011	30,172	26,890	27,239	25,472
6	223,159	21,835	25,646	25,704	25,468	23,633
7	71,340	23,871	38,566	27,937	26,245	24,511
8	69,276	22,948	40,349	27,006	26,953	25,165
9	57,291	23,852	41,001	27,166	25,702	24,100
10	65,647	22,734	41,108	26,390	25,711	24,024
11	27,080	27,716	55,538	29,779	26,168	24,337
12	29,816	29,901	67,557	32,683	26,283	24,528
13	27,914	26,107	54,643	28,286	26,298	24,569

PARTITION STEPS

DATA SETS	QUICKSORT	QUICKERSORT	BSORT	QSORTE	QSORTEM	QSORTEMI
1	1,999	1,023	1	1	1	1
2	1,999	1,405	1	1	1	1
3	1,999	1,999	2	505	490	234
4	1,023	1,023	1	1	1	1
5	1,462	1,336	296	874	782	316
6	1,527	1,384	131	781	663	284
7	1,530	1,351	577	870	808	283
8	1,555	1,350	574	929	788	289
9	1,425	1,333	611	899	832	276
10	1,442	1,345	615	941	802	287
11	1,337	1,331	767	995	837	282
12	1,338	1,345	761	996	840	295
13	1,345	1,327	769	983	828	282

CPU (SECONDS)

DATA SETS	QUICKSORT	QUICKERSORT	BSORT	QSORTE	QSORTEM	QSORTEMI
1	224.74	2.63	0.56	0.55	0.53	0.53
2	233.29	3.38	0.88	0.67	0.65	0.65
3	228.05	233.63	1.10	3.35	3.45	3.22
4	3.63	3.79	0.58	0.58	0.54	0.54
5	28.44	3.41	5.71	3.94	3.86	3.56
6	26.66	3.46	4.26	3.80	3.64	3.34
7	9.06	3.69	8.39	4.14	3.86	3.59
8	8.84	3.62	8.63	4.12	4.00	3.71
9	7.45	3.70	9.00	4.12	3.86	3.59
10	8.50	3.61	9.05	4.18	3.89	3.61
11	4.05	4.33	12.18	4.74	4.16	3.88
12	4.38	4.59	13.97	5.12	4.17	3.90
13	4.13	4.15	12.05	4.55	4.18	3.91

Qsorte does not have a worst case situation due to sorted subfiles like Quicksort or any of Quicksorts many improvments! Furthermore, many of the improvements that have been developed over the years for Quicksort may be used with Qsorte for even better performance. Qsortem and Qsortemi are two examples of improving the performance of Qsorte that we presented. Our results show Qsorte is a superior algorithm to Bsort. C.P.U. times run about the same for sorted and nearly sorted files (extremely fast), but for random files Qsorte is approximately three times faster. Bsort represents our first attempt at developing an algorithm that eliminates the worse case situation by providing an early exit for sorted subfiles. Qsorte represents a simpler and more direct approach than Bsort in determining if a subfile is in sorted order. It is a simpler algorithm and much easier to follow. Finally, the excellent test results exhibited by Qsorte suggests we should include the early exit checks for sorted subifles in any future improvements to Quicksort and we should no longer refer to the Quicksort algorithm as having a worst case behavior for sorted subfiles.

APPENDIX

THE QSORTE ALGORITHM

The following is a Pascal representation of the Qsorte algorithm. k[m],..., k[n] are the keys to be sorted and PIVOT is the value of the middle key. I and j are used to partition the subfiles so that at any time k[i] < PIVOT and k[j] > PIVOT. Lsorted is true whenever the left subfile is in sorted order, and rsorted is true whenever the right subfile is in sorted order. Flag is false when the partitioning process is completed.

```
procedure qsorte(m,n:integer; pivot_loc:integer);

  var     flag:             boolean;
          t,pivot:          real4;
          i,j,size:         integer;
          lsorted,rsorted:  boolean;

begin
if m<n then
  begin
  pivot:= k[pivot_loc];
  i := m-1; j := n+1; flag := true;
  lsorted := true; rsorted := true;

  while(flag) do
    begin
    i:=i+1;
    while  k[i] < pivot  do
      begin
      {determine if the left subfile}
      {is still sorted}
      if lsorted then
         if i>m then
            if k[i]<k[i-1] then lsorted:=false;
      i:=i+1;
      end;

    j:=j-1;
    while (k[j] >= pivot) and (j>=m) do
       begin
       {determine if the right subfile}
       {is still sorted}
       if rsorted then
          if j<n then
             if k[j]>k[j+1] then rsorted:=false;
       j:=j-1;
       end;

    if i<j then
       begin t:=k[j]; k[j]:=k[i]; k[i]:=t;
       {swap k[i] and k[j]}
       {check if the pivot value was moved}
       if i=pivot_loc then pivot_loc:=j;
       {determine if the left subfile}
       {is still sorted}
       if lsorted then
          if i>m then
             if k[i]<k[i-1] then lsorted:=false;
       {determine if the right subfile}
       {is still sorted}
       if rsorted then
          if j<n then
             if k[j]>k[j+1] then rsorted:=false;
       end
       else
       flag := false;
  end; {of while flag loop}

    {if the right subfile is not sorted then}
    {swap the pivot value with k[i].        }
    {this also takes care of the            }
    {special case for an empty left subfile }
    if not rsorted then
       begin
       {swap  k[i] and k[pivot_loc] }
       t:=k[i]; k[i]:=k[pivot_loc];
       k[pivot_loc]:=t;
       i:=i+1;
       end;

    if not lsorted then
       begin
       size:=j-m+1;
       if size>2
          then qsorte(m,j, (m+j) div 2)
          else if size=2 then
                  if k[m]>k[m+1] then
                  begin
                  t:=k[m]; k[m]:=k[m+1];
                  k[m+1]:=t;
                  end;
       end;

    if not rsorted then
       begin
       size:=n-i+1;
       if size>2
          then qsorte(i,n, (i+n) div 2)
          else if size=2 then
                  if k[n]<k[n-1] then
                  begin
                  t:=k[n]; k[n]:=k[n-1];
                  k[n-1]:=t;
                  end;
       end;

  end; {of m<n}

end; {of qsorte}
```

REFERENCES

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. Data Structures and Algorithms. Addison-Wesley, Reading, Mass., 1983. An excellent undergraduate textbook on data structures and algorithm analysis.

2. Bently, J.L. Programming Pearls-How to Sort. Commun. ACM 27, 4 (Apr. 1984), 287-291. This article presents several sorting algorithms that are simple to understand and easy to implement. Programming Pearls is an excellent column that presents in an entertaining manner practical solutions to "everyday" problems in computing.

3. Cook, C.R., and Kim, D.J. Best sorting algorithm for nearly sorted lists. Commun. ACM 23, 11 (Nov. 1980), 620-624. An excellent article describing a new sorting algorithm for nearly sorted lists. The algorithm is a combination Straight Insertion sort and Quicksort with merging. Results of an empirical study comparing this algorithm with Straight Insertion sort, Quickersort, Shellsort, Mergesort, and Heapsort are given.

4. Hibbard, T.N. An Empirical study of minimal storage sorting. Commun. ACM 6, 5 (May 1963), 206-213. Four sorting algorithms are developed and analyzed when storage is limited. A minimal storage Quicksort algorithm will always sort the shorter subsequence first.

5. Hillmore, J.S. Certification of Algorithms 63, 64, 65. Commun. ACM 5, 8 (Aug. 1962), 439. A Quicksort algorithm is presented that sorts subfiles of size three or more.

6. Hoare, C.A.R. Algorithm 64: Quicksort. Commun. ACM 4, 7 (July 1961), 321. A sorting algorithm is presented called Quicksort.

7. Hoare, C.A.R. Quicksort. Computer T. 5, 4 (Apr. 1962), 10-15. Some modifications to his original Quicksort algorithm are presented. Small subfiles should be sorted by some other sorting algorithm such as insertion sort.

8. Knuth, D.E. The Art of Computer Programming. Vol.3, Sorting and Searching. Addison-Wesley, Reading, Mass., 1972. The complete reference book on sorting and searching.

9. Loeser, R. Some performance tests of "quicksort" and "descendents." Commun. ACM 17, 3 (Mar. 1974), 143-152. The article reports results of an empirical study comparing Quicksort, two of its descendents (Quickersort and qsort), Shellsort, Stringsort, and Treesort.

10. Motzkin, D. Meansort. Commun. ACM 26, 4 (Apr. 1983), 250-251. This article describes a sorting algorithm based on Quicksort called Meansort where the mean value of each file controls the partitioning process. Results of an empirical study comparing Meansort and Quicksort are presented. Futher information on this article appears in the Technical Correspondence section of Commun. ACM 27, 7 (July 1984), 719-722.

11. Scowen, R.S. Algorithms 271: Quickersort. Commun. ACM 8, 11 (Nov. 1965), 669-670. An algorithm is presented called Quickersort where the middle element is choosen as the pivot value. He also suggests processing smaller subfiles first and not to split subfiles of size two or less.

12. Sedgewick, R. Implementing Quicksort Programs. Commun. ACM 21, 10 (Oct. 1978), 847-857. This is an excellent article on the implementation of Quicksort algorithms and many of its variations. It presents an overview of the properties of Quicksort.

13. Sedgewick, R. Algorithms. Addison-Wesley, Reading, Mass., 1983. An excellent reference book on algorithms. Chapter nine deals with Quicksort.

14. Singleton, R.C. Algorithm 347: An Efficient Algorithm for Sorting with Minimal Storage. Commun. ACM 12, 3 (Mar. 1969), 186-187. A modification to Quicksort is presented where a median-of-three method is used to determine the pivot value.

15. Standish, T.A. Data Structure Techniques. Addison-Wesley, Reading, Mass., 1980. An excellent textbook on data structures and algorithm for first-year graduate or advanced undergraduate students.

16. Van Emden, M.N. Algorithm 402: Increasing the Efficiency of Quicksort. Commun. ACM 13, 11 (Nov. 1970), 693-694. An algorithm called Qsort is presented where two pivot values are used.

17. Wainwright, R.L. A Class of Sorting Algorithms Based on Quicksort. Commun. ACM 28, 4 (Apr. 1985), 396-402. Three algorithms are presented called Bsort, Xsort and Ysort each minor variations of each other. The algorithms modify Quicksort in such a way that sorted subfiles are detected and not partitioned any further. Further information on this article appears in the Technical Correspondence section of Commun. ACM 29, 4 (April 1986), 331-335.

Author's Present Address: Roger L. Wainwright, Computer Science Dept., The University of Tulsa, 600 South College Avenue, Tulsa, OK 74104.