Edgar H. Sibley
Panel Editor

Bsort, a variation of Quicksort, combines the interchange technique used in Bubble sort with the Quicksort algorithm to improve the average behavior of Quicksort and eliminate the worst case situation of $O(n^2)$ *comparisons for sorted or nearly sorted lists. Bsort works best for nearly sorted lists or nearly sorted in reverse.*

A CLASS OF SORTING ALGORITHMS BASED ON QUICKSORT

ROGER L. WAINWRIGHT

Although there is no internal sorting algorithm that is best for every situation, the Quicksort algorithm introduced by Hoare [2, 3] is widely accepted as the most efficient internal sorting technique. Quicksort sorts a list of keys $A[1], A[2], \ldots, A[n]$ recursively by choosing a key m 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 j, $A[1], A[2], \ldots, A[j]$ contain all the keys with values less than m, and $A[j+1], A[j+2], \ldots, A[n]$ contain all the keys with values greater than or equal to m. The elements $A[1], A[2], \ldots, A[j]$ are called the left sublist, and the elements $A[j+1], A[j+2], \ldots, A[n]$ 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.

There have been many improvements and modifications to Quicksort (e.g., Quickersort [9] and qsort [12]). Whereas Quicksort partitions the original list using the *first* element as an estimate of the median, Quickersort partitions using the *middle list element* as an estimate of the median. Some additional descendants of Quicksort may be found in [4–7, 10, 11], and a more recent variation, Meansort, has been proposed by Motzkin [8]. Meansort does not select any particular key to control the partitioning process; instead, the partitioning is determined by the mean value of each list. During the first pass, the first key is selected as the key controlling the partitioning process. While the keys are being compared and interchanged, the values of the keys of each subfile are summed. When the partitioning process is finished, the two means are computed. Each subfile from then on is partitioned based on its mean, and new mean values are computed at each partitioning step. For many distributions of keys, the mean value will be a better estimate of the median value than a key selected at random. Furthermore, the distribution of the keys does not affect the performance of Meansort. Meansort is one of several algorithms (see p. 398) that will be compared later in this paper.

However, the fundamental drawback of Quicksort and its variations remains the same: The worst-case behavior of $O(n^2)$ comparisons occurs when a list is nearly sorted or nearly sorted in reverse order, although it would seem that a completely sorted or nearly sorted list would take the least time to sort. Cook and Kim [1] have developed an excellent sorting algorithm for nearly sorted lists, which, for lack of any other nomenclature, we will refer to as CKsort. CKsort is a combination of Straight Insertion sort and Quickersort with merging and is designed specifically for nearly sorted lists. It performs poorly on any other distribution of keys including nearly sorted in reverse.

In CKsort, the original list is first scanned, and pairs of unordered elements are removed and placed in another array. After a pair of unordered elements has been removed, the next pair compared is the elements immediately preceding and immediately following the pair just removed. After repeating this for all elements in the original list, the array of unordered pairs of elements is sorted by Straight Insertion, if there are no more than 30 elements, and by Quickersort otherwise. Finally, the two sorted arrays are merged into one array.

Bsort, the technique presented in this paper, is a variation of Quicksort that is designed to work best for

nearly sorted lists as well as nearly sorted in reverse. For all distribution of keys within a file, Bsort requires no more than $O(n \log_2 n)$ comparisons. In fact, for sorted lists or lists sorted in reverse, the algorithm takes $O(n)$ comparisons. Bsort uses the technique associated with Bubble sort. In a Bubble sort, a check is made after each pass to determine if the list is sorted. If the list is sorted, the algorithm terminates and further passes are not necessary. As the keys are being compared and interchanged, it is easy to determine if the list is sorted or not after each pass has been completed. Similarly, after each pass in Bsort, each subfile of keys is checked to see if it is sorted. If it is, the subfile need not be partitioned again. While the keys are being compared and interchanged during the construction of each subfile, it is easy to determine if the subfile is in sorted order or not.

It will be shown that Bsort has no "worst case" behavior, that it works best for nearly sorted lists, and on an overall basis improves the average performance of Quicksort.

THE ALGORITHM FOR BSORT

As in Quickersort, Bsort selects the middle key during each pass as the key 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 unneeded. This is the major advantage of Bsort over Quicksort. 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, which represents a second advantage over Quicksort (i.e., fewer partition steps). 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. The following example illustrates how the Bsort algorithm works. (A Pascal version of the Bsort algorithm is given in the Appendix.)

Consider the following file with the middle key, 35, as the pivot key controlling the partitioning process. We start with two pointers, i and j, pointing to the left- and rightmost keys in the file, respectively.

91 62 3 84 35 10 97 28 **49**
i j

First we attempt to move i to the right until a number larger than or equal to the pivot value is found. In this case, since i rests on the value 91, i does not change and remains on 91. Next we move j to the left until a number less than the pivot value is found. This yields the following:

91 62 3 84 35 10 97 **28** 49.
i j

Exchange the keys pointed to by i and j.

28 62 3 84 35 10 97 **91** 49
i j

Now 91 and 49 are exchanged to preserve the property that the leftmost value in the right subfile is always the smallest value of that subfile. Since an exchange has occurred within the right subfile, the subfile will not necessarily be in sorted order when the partitioning process is completed.

28 62 3 84 35 10 97 **49** 91
i j

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 62. Move j left until it rests on a value less than the pivot (in this case 10), and note that as j passes over 97, values 49 and 97 are exchanged preserving the leftmost value in the right subfile as the smallest. This yields

28 **62** 3 84 35 **10** 49 97 91.
i j

We exchange the keys pointed to by i and j yielding

28 **10** 3 84 35 **62** 49 97 91,
i j

and we preserve the rightmost key of the left subfile as the largest and the leftmost key of the right subfile as the smallest by exchanging 28 with 10 and 62 with 49. Since an exchange has now occurred in the left subfile, it will not necessarily be in sorted order when the partitioning is completed. This yields

10 **28** 3 84 35 **49** 62 97 91.
i j

Finally, move i right to 84, exchanging 28 with 3 along the way, and move j left also to 84 yielding

10 3 28 **84** 35 49 62 97 91.
ij

The process terminates when i and j meet. The key pointed to by i and j must now be placed into the proper subfile. Since 84 is larger than the pivot, it belongs in the right subfile, and 35 and 84 are exchanged yielding the final partition:

[10 3 28] [35 84 49 62 97 91].

Any subfile of length three can be placed in order by

one comparison and possibly one exchange; thus, after exchanging 10 with 3, the left subfile is sorted. The original problem has now been reduced to sorting the file

[84 49 62 97 91].

VARIATIONS TO BSORT

The Bsort technique of combining Quicksort with the Bubble sort technique represents a new class of Quicksort algorithms. Many variations may be applied to the basic Bsort algorithm; listed below are three such variations: BMsort, Xsort, and Ysort. Hereafter, we will refer to the four algorithms Bsort, BMsort, Xsort, and Ysort collectively as a class of algorithms called BSORT algorithms.

BMsort. BMsort is a combination of Meansort and Bsort; it uses the mean value of a subfile to control the partitioning process rather than the middle key.

Xsort. Xsort proceeds along the same lines as Bsort except that during the construction of the left subfile, the location of the minimum key value is recorded, and during the construction of the right subfile, the location of the maximum key value is recorded. When the partitioning process is completed, the minimum key in the left subfile is interchanged with the leftmost key of that subfile, and the maximum key in the right subfile is interchanged with the rightmost key of that subfile. Therefore, each subfile will have the minimum and maximum keys located at the left and right ends, respectively. If a subfile has three keys or less, it is in fact sorted by the construction of the subfile. In Xsort, if a subfile has four keys, it can be sorted by one comparison and at most one interchange; a subfile will not be partitioned further unless it has at least five keys. Thus, this algorithm reduces even further the number of partition steps required. It is possible to combine Xsort with Meansort, but empirical results indicate this does not improve the performance of Xsort.

Ysort. Ysort proceeds as in traditional Quicksort. There is no interchange of keys during the construction of the subfiles as in Bsort. Instead, the locations of the maximum and minimum key values are recorded for each subfile as the subfiles are constructed. After the partitioning process is completed, the minimum and maximum keys for each subfile are interchanged with the leftmost and rightmost keys, respectively, in the subfiles. The Ysort algorithm achieves the same end result as Xsort; namely, the minimum and maximum keys in each subfile are located at the left and right

TABLE I. Comparison of Sorting Algorithms on 200 Elements

	Data Sets										
	1	2	3	4	5	6	7	8	9	10	11
Interchanges											
Quicksort	199	199	662	196	236	208	276	364	367	369	371
CKsort	0	199	0	5	228	32	262	362	379	363	375
Meansort	0	100	0	69	201	167	209	390	366	382	372
BMsort	0	100	0	69	201	593	479	1,091	1,140	1,050	1,142
Bsort	0	100	0	69	201	597	481	1,406	1,762	1,190	1,534
Xsort	0	100	0	69	201	577	479	1,510	1,188	1,220	1,264
Ysort	0	100	0	69	201	161	207	342	324	334	324
Comparisons											
Quicksort	20,298	20,298	1,324	16,096	13,646	5,844	6,874	1,820	2,009	1,775	1,856
CKsort	199	30,498	199	616	20,582	786	15,170	7,015	5,418	4,782	3,770
Meansort	1,743	1,943	201	1,881	2,145	2,077	2,161	2,546	2,570	2,526	2,559
BMsort	400	600	401	942	1,206	2,036	1,979	2,745	2,886	2,695	2,810
Bsort	400	600	401	944	1,206	2,214	2,000	3,533	4,348	3,160	3,978
Xsort	597	797	599	1,382	1,689	2,782	2,607	4,916	3,872	4,023	4,082
Ysort	400	600	401	2,056	2,504	2,621	2,588	3,494	4,310	3,647	4,059
Partition steps											
Quicksort	199	199	127	175	178	149	160	133	131	133	135
CKsort	0	199	0	5	191	19	161	135	132	130	136
Meansort	199	199	1	199	199	199	199	199	199	199	199
BMsort	1	1	1	5	7	41	28	65	68	62	63
Bsort	1	1	1	5	7	54	28	69	75	82	79
Xsort	1	1	1	5	9	38	24	51	53	52	50
Ysort	1	1	1	34	40	45	41	50	45	49	53
CPU (seconds)											
Quicksort	2.40	2.51	0.33	1.93	1.73	0.81	0.94	0.35	0.38	0.35	0.36
CKsort	0.11	10.79	0.11	0.33	7.68	0.40	7.14	2.33	3.32	2.74	1.95
Meansort	3.51	3.59	0.39	3.59	3.63	3.61	3.62	3.75	3.85	3.73	3.81
BMsort	0.13	0.16	0.13	0.61	0.71	1.59	1.43	2.01	2.15	1.97	2.06
Bsort	0.13	0.16	0.13	0.22	0.28	0.66	0.49	0.99	1.10	0.96	1.06
Xsort	0.15	0.19	0.15	0.28	0.34	0.65	0.55	1.06	0.91	0.93	0.93
Ysort	0.13	0.17	0.14	0.53	0.62	0.65	0.63	0.79	0.86	0.80	0.88

ends, respectively. Furthermore, this is accomplished without the added expense of key interchanges during the construction of the subfiles.

In so doing, however, we lose the ability to tell if a subfile is sorted or not! This problem is remedied in the following way: In the construction of the left subfile, if the new key placed into the left subfile *always* becomes the new *maximum* key of the subfile, then the subfile is sorted upon completion of the partitioning process. Similarly, during construction of the right subfile, if the new key placed into the subfile *always* becomes the new *minimum* key of the subfile, then the subfile is sorted upon completion of the partitioning. This is easy to determine since the minimum and maximum keys have already been processed. Ysort thereby achieves exactly the same end result as Xsort but by a totally different algorithm.

EMPIRICAL TEST RESULTS

Test results for the following seven sorting algorithms—Quicksort, CKsort, Meansort, BMsort, Bsort, Xsort, and Ysort—are compared and discussed below.

Table I gives experimental results for the seven algorithms acting on data sets of 200 elements, and Table II for the same seven algorithms on data sets of 2000 elements. The tables show 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 small 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.

In each table, data set 1 is an ordered list, and data set 2 is an ordered list in reverse order. Data set 3 is composed of numbers of the same value. Data sets 4 and 5 consist of lists with a sortedness ratio of 2 percent and 2 percent in reverse order, respectively. Data sets 6 and 7 are lists with a sortedness ratio of 8 percent and 8 percent in reverse order, respectively, while data sets 8–11 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

TABLE II. Comparison of Sorting Algorithms on 2000 Elements

	Data Sets										
	1	2	3	4	5	6	7	8	9	10	11
Interchanges											
Quicksort	1,999	1,999	9,893	2,441	3,285	3,304	4,155	5,175	5,183	5,207	5,232
CKsort	0	—	0	99	—	547	—	5,344	5,181	5,266	5,171
Meansort	0	1,000	0	1,923	2,895	2,951	3,645	5,437	5,388	5,494	5,541
BMsort	0	1,000	0	7,613	9,333	13,567	13,287	18,929	18,670	19,006	18,937
Bsort	0	1,000	0	7,785	9,383	14,589	13,137	24,258	23,370	23,896	22,199
Xsort	0	1,000	0	7,663	9,229	13,445	13,209	23,645	21,700	21,824	22,491
Ysort	0	1,000	0	1,727	2,651	2,661	3,307	4,783	4,766	4,868	4,835
Comparisons											
Quicksort	2,002,998	2,002,998	19,786	440,347	309,344	81,907	58,668	27,958	27,164	28,342	28,239
CKsort	1,999	—	1,999	6,757	—	10,168	—	123,770	97,585	97,550	87,630
Meansort	23,951	25,951	2,001	27,797	29,741	29,853	31,241	35,319	34,993	35,379	35,391
BMsort	4,000	6,000	4,001	29,351	32,181	36,869	36,933	43,924	43,378	43,900	43,822
Bsort	4,000	6,000	4,001	30,536	33,294	42,168	39,682	56,386	54,654	55,644	52,304
Xsort	5,997	7,997	5,999	42,387	45,693	53,278	52,274	73,102	66,992	67,242	69,071
Ysort	4,000	6,000	4,001	41,796	43,360	49,626	50,309	66,719	65,138	63,216	63,594
Partition steps											
Quicksort	1,999	1,999	1,023	1,538	1,483	1,502	1,499	1,332	1,336	1,342	1,332
CKsort	0	—	0	53	—	213	—	1,341	1,341	1,313	1,344
Meansort	1,999	1,999	1	1,999	1,999	1,999	1,999	1,999	1,999	1,999	1,999
BMsort	1	1	1	259	282	442	423	667	629	649	660
Bsort	1	1	1	318	385	601	582	760	774	772	743
Xsort	1	1	1	243	310	449	433	512	504	502	508
Ysort	1	1	1	427	431	453	486	498	499	498	505
CPU (seconds)											
Quicksort	221.73	233.36	3.63	50.95	36.82	10.01	7.58	4.15	4.04	4.21	4.18
CKsort	0.40	—	0.41	2.65	—	4.63	—	81.84	60.30	60.25	52.13
Meansort	45.21	45.54	3.31	45.83	46.14	45.96	46.38	48.02	47.63	47.98	47.85
BMsort	0.56	0.87	0.58	22.84	23.95	27.96	27.68	31.70	31.10	31.57	31.47
Bsort	0.56	0.87	0.58	5.84	6.62	9.12	8.64	12.27	12.07	12.22	11.57
Xsort	0.80	1.11	0.83	8.75	7.87	9.86	8.88	13.38	12.42	12.45	12.75
Ysort	0.63	0.95	0.66	8.69	7.90	8.69	8.99	10.97	10.79	10.57	10.65

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 [1].

As shown in Tables I and II, Meansort compares favorably to the class of BSORT algorithms (Bsort, BMsort, Xsort, and Ysort) and even superior to Quicksort when comparing interchanges, comparisons, and partition steps. Also, the sorting time of Meansort appears to be independent of the order of the keys; instead, it depends only on the distribution of their values (see CPU times for Meansort), which are the same observations reported by Motzkin. Based on the number of interchanges, comparisons, and partition steps, Motzkin concludes that Meansort is a considerable improvement over standard Quicksort and has a better average behavior than standard Quicksort [8]. Our results show Meansort to have very poor CPU times compared to the other algorithms tested. The additional CPU time required to determine the mean value for each partition seems to outweigh the advantage.

Among the class of BSORT algorithms, BMsort had the worst CPU times. This is undoubtedly due to the additional calculations required to determine the mean key value in each partition. Bsort, Xsort, and Ysort algorithms are minor variations of each other and, as expected, yield similar CPU times. However, Xsort on an overall basis yielded slightly higher CPU times than both Bsort and Ysort over all the data sets tested. Of the seven algorithms tested, Ysort consistently yielded the minimum number of interchanges over all data sets. This is because it does not perform a "bubble type" interchange during the partitioning process, although Ysort still determines if a subfile is sorted or not.

As a class, the BSORT algorithms yielded the minimum number of partitions required to sort. This is what the BSORT algorithms were designed to do, that is, test for sorted partitions and prevent further partitioning. Ysort yielded the minimum number of partitions among random data sets, whereas Xsort yielded the minimum number of partitions for nearly sorted data sets or nearly sorted in reverse. The "bubble interchanges" prove invaluable for nearly sorted lists (see Xsort and Bsort partition steps for nearly sorted lists in both tables). Each key tends to move toward its final position in a nearly sorted list quicker when bubble interchanges are done than when the interchanges are not done, thereby greatly reducing the number of partitions.

SUMMARY AND CONCLUSIONS

CKsort is still the best algorithm for nearly sorted lists. Although CKsort was not designed for other than nearly sorted lists, we included performance on other data sets for the sake of completeness. Dash entries indicate a worst case situation where calculations were not made. Quicksort is still the best algorithm for random keys. In the worst case situation of nearly sorted lists or nearly sorted in reverse, the performance of Quicksort is extremely poor. Meansort proved to be the worst of the seven algorithms tested with respect to CPU time, and BMsort was worst by far among the class of BSORT algorithms. Results show that both Bsort and Ysort algorithms are considerable improvements over Quicksort, CKsort, and Meansort: Bsort and Ysort have better average behavior and no worst case situations like Quicksort and CKsort. Instead, they have a best case situation for nearly sorted lists or nearly sorted in reverse. Bsort and Ysort also performed reasonably well on random keys and are particularly efficient in situations where there are repeating keys.

APPENDIX

THE BSORT ALGORITHM

Following is a Pascal representation of the Bsort algorithm. $K[m], \ldots, k[n]$ are the keys to be sorted, and mid_key is the value of the middle key. I and j are used to partition the subfiles so that at any time $k[i] <$ mid_key and $k[j] \geq$ mid_key. Left_flag is true whenever the left subfile is not in sorted order, and right_flag is true whenever the right subfile is not in sorted order. Flag is false when the partitioning processes are completed.

```
procedure Bsort (m,n:integer; mid_key:real);
var flag,left_flag,right_flag:boolean;
    t:real;
    i,j,size:integer;

begin
if m<n then
```

```
begin
left_flag:=false; right_flag:=false;
i:=m; j:=n; flag:=true;

while(flag) do
  begin

  while (k[i]<mid_key) and (i<>j) do
    begin                    {build the left subfile ensuring}
    if i<>m then             {that the rightmost key is      }
       if k[i-1]>k[i] then   {always the largest of the      }
          begin              {subfile                        }
          t:=k[i-1]; k[i-1]:=k[i]; k[i]:=t;
          left_flag:=true;
          end
    i:=i+1;
    end;

  while (k[j]>=mid_key) and (i<>j) do
    begin                    {build the right subfile ensuring}
    if j<>n  then            {that the leftmost key is       }
       if k[j]>k[j+1] then   {always the smallest of the     }
          begin              {subfile                        }
          t:=k[j+1]; k[j+1]:=k[j]; k[j]:=t;
          right_flag:=true;
          end;
    j:=j-1;
    end;

  if i<>j then
     {interchange k[i] from the left subfile}
     {with k[j] from the right subfile      }
     begin t:=k[j]; k[j]:=k[i]; k[i]:=t;
     end
     else
     begin {i=j}
     {partitioning into left and right subfiles has been completed}
     if k[j]>=mid_key then
        begin
        {check the right subfile to ensure       }
        {the first element, k[j], is the smallest}
        if k[j]>k[j+1] then
           begin
           t:=k[j+1]; k[j+1]:=k[j]; k[j]:=t;
           right_flag:=true;
           end;
        end
        else
        begin
        {check the left subfile to ensure}
        {the last element, k[i-1], is    }
        {the largest                     }
        if k[i-1]>k[i] then
           begin
           t:=k[i-1]; k[i-1]:=k[i]; k[i]:=t;
           left_flag:=true;
           end;
        if k[i-2]>k[i-1] then
           begin
           t:=k[i-1]; k[i-1]:=k[i-2]; k[i-2]:=t;
           end;
```

```
            end;
            flag:=false;
         end;
      end; {of while flag loop}

   {process the left subfile}
   size:=i-m;
   if size>2 then          {subfile must have at least 3 elements}
      if left_flag then    {to process and not already sorted.    }
         if size=3 then    {special case of 3 elements            }
            begin          {place k[m] and k[m+1] in sorted order}
            if k[m]>k[m+1] then
                 begin
                 t:=k[m+1]; k[m+1]:=k[m]; k[m]:=t;
                 end;
            end
            else
            Bsort(m,i-2,k[trunc((m+i-2)/2)]);

   {process the right subfile}
   size:=n-j+1;
   if size>2 then          {subfile must have at least 3 elements  }
      if right_flag then   {to process and not already sorted.     }
         if size=3 then    {special case of 3 elements             }
            begin          {place k[j+1] and k[j+2] in sorted order}
            if k[j+1]>k[j+2] then
                 begin
                 t:=k[j+1]; k[j+1]:=k[j+2]; k[j+2]:=t;
                 end;
            end
            else
            Bsort(j+1,n,k[trunc((j+1+n)/2)]);

   end {of m<n}
end; {of procedure Bsort}
```


REFERENCES

1. 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, Merge sort, and Heapsort are given.
2. Hoare, C.A.R. Algorithm 64: Quicksort. *Commun. ACM* 4, 7 (July 1961), 321.
3. Hoare, C.A.R. Quicksort. *Computer T.* 5, 4 (Apr. 1962), 10–15.
4. 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.
5. Loeser, R. Some performance tests of "quicksort" and "descendents." *Commun. ACM* 17, 3 (Mar. 1974), 143–152. This article reports results of an empirical study comparing Quicksort, two of its descendents (Quickersort and qsort), Shellsort, Stringsort, and Treesort.
6. Loeser, R. Survey on Algorithms 347, 426, and Quicksort. *ACM Trans. Math. Softw.* 2, 3 (Sept. 1976), 290–299. This is a continuation of his previous work [5]. Three additional sorting algorithms are compared: Sinsort, Richsort, and Bronsort.
7. Motzkin, D. A stable Quicksort. *Softw. Pract. Exper.* 11, 6 (June 1981), 607–611.
8. 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. Further information on this article appears in the Technical Correspondence section of *Commun. ACM* 27, 7 (July 1984), 719–722.
9. Scowen, R.S. Algorithm 271: Quickersort. *Commun. ACM 8,* 11 (Nov. 1965), 669–670.
10. Sedgewick, R. Quicksort with equal keys. *SIAM J. Comput. 6,* 2 (June 1977), 240–267.
11. Singleton, R.C. Algorithm 347: An efficient algorithm for sorting with minimal storage. *Commun. ACM 12,* 3 (Mar. 1969), 185–187.
12. vanEmden, M.N. Algorithm 402: Increasing the efficiency of Quicksort. *Commun. ACM* 13, 11 (Nov. 1970), 693–694.

CR Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sorting and searching*

General Terms: Algorithms

Additional Key Words and Phrases: sorting, Quicksort, efficient sorting, Bubble sort

Received 4/84; accepted 9/84

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