An Internal Hybrid Sort Algorithm Revisited

O. NEVALAINEN AND T. RAITA
Department of Mathematical Sciences, Computer Science, University of Turku, SF-20500 Turku, Finland

Two hybrid methods of distributive sort and quicksort are given. The first method sorts an array of records and the second one sorts a linearly linked list in a stable way. The expected running time of the methods is O(n) for n records and for a wide class of distributions of the keys (including all bounded densities with a compact support). For most other distributions the running time is O(n log n), and the worst case time is O(n^2). The array version needs extra storage space for n records and approximately n/5 integers. In the linked list version only an array of n/5 pointers is needed. The observed running times of the algorithms compare favourably with those of other efficient bucket sort algorithms.

Received March 1988, revised August 1991

1. INTRODUCTION

Internal sorting algorithms presented in this paper are representatives of a hybrid method which consists of a distribution phase followed by an application of a comparison based sorting algorithm. The former scatters the key values into buckets which are thereafter sorted by the latter technique. The results are then gathered to form the sorted sequence of records. A number of different variants of this method has been proposed[1-4, 7, 9-13] and it seems that the implementation of a hybrid of comparison and distribution sort is not as clearcut as one might expect. There are several reasons for this.

First, in contrast to quicksort or heapsort, the hybrid is no longer an *in situ* method. This is due to the fact that the records are grouped to form a number of buckets and the information for this requires $O(n)$ extra storage. The actual observed running time depends strongly on the amount of extra storage used.[7] Representing the buckets with linked lists and using an auxiliary array for the records leads to very efficient sorting. On the other hand, if memory requirements are strict, only an extra array of m integers is sufficient, where m is the number of buckets used. Secondly, the performance of the hybrid method should be comparable to the best comparison based sorts for ill-behaved distributions of the key values. Finally, to be fair to the comparison based sorting, the bucket sort may not have any prior knowledge of the key values. For example, we may not assume that the minimal key value is known, that the keys are not identical, or that the sentinel element is in its correct place. Although all these details are easy to consider, the determination takes processing time which should not be ignored. The best sort routines based on comparisons are so fast that each extra operation in bucket sort decreases the margin between the running times of the two methods.

In this paper we give two general-purpose internal bucket sort programs, *Dsort* and *Ldsort* which are

(1) theoretically strong in the sense that
  - they have an $O(n)$ expected running time for the key distributions occurring frequently in practical situations[6], an $O(n \log n)$ expected running time for most other distributions and an $O(n^2)$ worst case running time in extremely improbable cases,
  - their demand for the extra memory space is moderate;

(2) applicable in practical situations because they
  - sort quickly in commonly occurring cases, and
  - do not cause unexpectedly long running times for special types of key sets.

Dsort is a refinement of *D-2*[12] (written in FORTRAN). The algorithm makes a compromise between the amount of storage space and the running time. No explicit linking is needed in the bucketing but an extra copy of the records is utilized along with a counter array of length of m. G. H. Gonnet[8] gives a bucket sort routine, interpolation sort (written in C), with the same demand for storage space. The code is very compact and the program is fast for uniformly distributed key sets. However, the values of the minimal and maximal keys are not determined in the routine and the processing degenerates to $O(n^2)$ insertion sort when all but one key value differs significantly from all the other values.

Other related methods are *D-3*[12] and *Usort*.[3] Both use explicit (cursor) linking of the buckets and are very fast for smooth key distributions. A two-stage distribution function is used in the method *DDP* (double distributive partitioning) of M. T. Noga and D. C. S. Allison.[13] Also in *DDP* the linking is explicit by cursors. An even distribution of bucket sizes is searched by a two-phase procedure. This results in a theoretically appealing method and we are interested in knowing whether its more involved bucket boundary calculations are valuable in practical sorting situations, too.

Ldsort is related to *Dsort* but it manipulates a linearly linked list of records. Now we need as extra storage only a header array of m pointers to access the bucket lists. The use of linking presupposes an efficient comparison based sorting of a linked structure. It turns out that by using a suitable technique and a careful coding, the sorting can be done in a stable way, i.e. in such a way that the initial mutual order of the records with equal keys is preserved in the process. The resulting sorting algorithm is logically closely related to *Usort*. However, the true linking by pointers (as compared to the cursor links in *Usort*) and the actions to prevent overwhelming processing time for exceptional key distributions make the new routine more robust. A linked version of *DDP* is also given for comparison.

The paper is organised as follows. Section 2 describes a Pascal implementation of *Dsort* in detail. Some observations on *Usort* and *DDP* are made in Section 3. In Section 4 we report a summary of the results of test runs with array based bucket sort programs and some other efficient sorting programs. The implementation of *Ldsort* and *LDDP* is discussed in Section 5. Test results for these methods are summarised in Section 6. Finally, Section 7 contains some concluding remarks.

2. IMPLEMENTATION

2.1 Preliminaries

The non-recursive bucket sort titled *Dsort* has a heading of the form:

```
procedure dsort (var p: array_to_be_sorted; lo, up: integer);
```

and the following declarations:

```
const
  large_file = 2000;
  max_m = 10000;
  ins_level = 9;
var
  i, j, m, imax: integer;
  counter: array [0..max_m] of integer;
  s: array_to_be_sorted;
  temp: element;
  a, b: real;
  min, max: key_type;
```

The procedure orders the elements of *p* from the index *lo* to index *up* to form an increasing sequence of key values.

(Note 1. We have restricted ourselves here to the integer type index values. Further, it would be more elegant to choose a conformant array schema to obtain a flexible interface to the procedure. However, this was not carried out because only few systems support it. Both these restrictions could be avoided by using a linked list for the records to be sorted and passing the head of the list to the sort routine.)

The constant *large_file* is used as a value for choosing the method of sorting, *ins_level* defines the threshold value which expresses when it is preferable to change from quicksort to insertion sort. Max_m+1 stands for the maximal number of buckets to be used. The *calling block* must define the type *array_to_be_sorted* as an array of records of type *element* containing the *keyfield* which is of *key_type*. For example:

```
const
  lowindex = 1;
  highindex = 30000;
type
  indexrange = lowindex..highindex;
  key_type = real;
  element = record
              keyfield: key_type;
              data: array [1..30] of char;
            end;
  array_to_be_sorted = array [indexrange] of element;
```

(Note 2. Due to the restrictions of Pascal, the type of records to be sorted is fixed, that is, *Dsort* must know the field name according to which the sorting is to be done. To increase the usability of the routine, we ought to be able to give the type of the record as a parameter. Then the parameter list would be augmented with a function which compares two records and yields a Boolean result expressing which one of the arguments is greater. This would generalise our procedure to the extent that it could be used to sort all records for which total ordering is defined. We implemented this kind of routine, too. The sorting was however significantly slowed down due to the large time used for comparing keys.)

2.2 Procedure body

The distributive method is unnecessarily complicated for small files. Therefore, if the number of items is less than *large_file* elements, quicksort is applied. Otherwise, a three-level hierarchy is used: the bucketing technique is applied as a preliminary step after which the buckets are sorted with quicksort. The quicksort procedure has a further threshold level: as soon as the partitioning results in a subset, the size of which is less than *ins_level*, the subset is left unsorted (partial quicksort). As a final step, the whole array is scanned using insertion sort to fix up the possible disorder which is clustered into small regions. The overall structure of the body of *Dsort* is:

```
{choose the method of sorting}
if up-lo + 1 < large_file then
  begin {small array}
    "use partial quicksort";
    "arrange p so that maximal element of the last
      subfile is in p[up]"
  end
else
  begin {large array}
    {initialisation}
    "Determine the minimum (min) and maximum (max)
      key values. Rearrange p so that p[up] is the maximal
      element and omit it from the sorting";
    if min < > max then
    begin
      "organise the records to form m buckets";
      "sort the buckets by partial quicksort"
    end
  end;
"insertionsort using a sentinel element at the rear";
```

Small array

Quicksort is applied directly to arrays containing at most *large_file* elements (typically *large_file* is of the order 2000). The routine *partial_quicks* is a modified version from that of Sedgewick[15]. The changes are:

(*a*) to make the $O(n^2)$ worst case improbable, the median-of-three is used for selecting the pivot element,

(*b*) to limit the stack space to $O(\log n)$, the tail recursion has been removed and the smaller subfile is always sorted first, and

(*c*) the insertion sort part at the end of Sedgewick's original quicksort procedure has been removed because we want to perform the insertion sort with a stopper element for the whole array containing all the partially sorted buckets.

Note that also in this branch of the program the array consists of 'buckets'. The upper limit for the size of a bucket is determined by *ins_level*.

The following code excerpt rearranges p so that the maximal element of the last subfile is in $p[up]$.

```
{Small array}
  partial_quicks(p, lo, up);
  max := p[up].keyfield; imax := up; j := up-ins_level;
  if j < lo then j := lo;
  for i := up-1 downto j do
    if p[i].keyfield > max then
      begin max := p[i].keyfield; imax := i end;
  swap(p[up], p[imax]);
  up := up-1;
```

The call of procedure *swap* is to be replaced in the fina code by the three statements:

```
temp := p[up]; p[up] := p[imax]; p[imax] := temp;
```

Large array
Initialisation

The records with the minimal and maximal keys are determined and the latter is moved to the correct place to act as a sentinel element in the insertion sort. The maximal element is excluded from sorting. Note that the modification of the upper limit *up* does not alter its value outside *Dsort*.

```
min := p[lo].keyfield; max := min; imax := lo;
for i := lo+1 to up do
    if p[i].keyfield < min
      then min := p[i].keyfield
      else if p[i].keyfield > max then
        begin max := p[i].keyfield; imax := i end;
swap(p[up], p[imax]);
up := up-1;
```

Organise the records to form *m* buckets

The substeps are
  S1: 'initialisation';
  S2: 'determination of the number of records in the buckets';
  S3: 'initialisation of the cursors to the buckets';
  S4: 'construction of the buckets';

S1: 'initialisation'
We divide the records into $m = n/k$ buckets. Here we use $k = 5$, which gave in test runs profitable sorting times for smooth key distributions. To maintain information about m buckets, we use $m+1$ storage locations from the array *counter*. At the beginning of the sorting the array is used to find the number of records in each bucket. Thereafter the same space serves for a set of cursors to the buckets. The value of m is restricted to *max_m*, which gives the high index of *counter*.

A record with a key x is placed in the bucket i ($0 \leqslant i \leqslant m-1$), where

$$i = floor[(x-min)*(m-1)/(max-min)]$$
$$= floor[x*a+b].$$

The coefficients a and b are given below.

```
m := trunc((up-lo+1)/5);
if m > max_m then m := max_m;
for j := 0 to m do counter[j] := 0;
a := (m-1)/(max-min);
b := -((m-1)*min)/(max-min);
```

S2: 'determination of the number of records in the buckets'
The number of records in each bucket is counted in *counter*[*i*] ($i = 0..m$). To speed up the construction of the buckets we store the elements of p temporarily into the array s. The step is as follows:

```
for i := lo to up do
  begin
    j := trunc(p[i].keyfield*a+b);
    counter[j] := counter[j]+1;
    s[i] := p[i]
  end;
```

S3: 'initialisation of the cursors to the buckets'
The counter array is transformed into a cursor array giving the address of the first free slot in each bucket.

```
            counter[0] := counter[0]+lo-1;
for j := 1 to m do counter[j] := counter[j-1]+counter[j];
```

Note that the bucket m is empty and therefore from step S1 we have $counter[m] = 0$. After S3 its value is identical with the value of *counter* $[m-1]$ ($= up-lo+1$). This is advantageous at the time quicksort is applied to the buckets.

S4: 'construction of the buckets'
The contents of s are mapped back to p to form the m buckets:

```
  for i := up downto lo do
    begin
      j := trunc(s[i].keyfield*a+b); p[counter[j]] := s[i];
        counter[j] := counter[j]-1;
    end;
```

after which the elements of the buckets reside in consecutive addresses in the original array.

Sort the buckets by partial quicksort

```
for j := m-1 downto 0 do
  partial_quicks (p, counter[j]+1, counter[j+1]);
```

Insertionsort
Because a sentinel element was stored in the distribution phase in $p[up+1]$ an efficient version of the insertion sort can be used. It is called by:

```
insert_stopper_max(p, lo, up);
```

2.3 Running time

The expected running time of bucketing methods has been analysed in ref. 2 (see also ref. 5). If an $O(n^2)$ algorithm is applied to the buckets and the keys $x_1, x_2, \ldots, x_n$ are from a distribution with the density f, the expected running time of the distributive algorithm is $O(n)$ for all bounded distributions. (See ref. 5, Theorem 1 for a very general class of distributions giving linear expected time.) On the other hand, Devroye and Klincsek showed that a necessary condition for the linearity is that f has a compact support (i.e. there exists a finite K such that $P(|X| > K) = 0$, where X has the density f). This result implies that all densities with an infinite tail, for example normal, gamma and chi-square densities, yield an over-linear running time. However, the keys can be modified using a non-linear order-preserving transformation so that the effect of the long tail is removed and the linearity is again reached. In order for this to be practical we should have information about the type of f so that we do not make the resulting density very skew. This feature has been omitted from *Dsort*.

In cases where the bucketing fails to distribute the keys into subsets of about the same size, the expected $O(n \log n)$ running time of quicksort is dominant. Still, the $O(n^2)$-worst case is possible in extremely special cases.[15]

3. *USORT* AND TWO-LEVEL BUCKET SORT

Usort is a variant of *Dsort* in which the link fields are stored explicitly in an array of n integers.[3, 7] This results in more straightforward code and avoids the expensive calculation of the bucket number for each element twice. The distributive phase and the reorganisation of the p array are done as follows:

```
{distributive phase}
  for i := 0 to m-1 do lhead[i] := 0;
  for i := lo to up do
    begin
      j := trunc(p[i].keyfield*a+b); link[i] := lhead[j];
      lhead[j] := i; s[i] := p[i];
    end;
{dump linked lists back to p}
  i := lo-1;
  for j := 0 to m-1 do
  if lhead[j] < > 0 then
    begin
      next := lhead[j]; i := i+1; j1 := i; p[i] := s[next];
      while link[next] < > 0 do
        begin
          next := link[next]; i := i+1; p[i] := s[next]
        end;
      partial_quicks (p, j1, i)
    end;
```

To get an understanding of the actual running time of *Dsort* in comparison with other bucket sort methods we implemented also the two-level bucket sort *DDP* by Noga and Allison.[13] Their nice algorithm uses 2 arrays of *ca.* $n/100$ integers and 2 arrays of around $n/100$ real numbers for collecting the statistics about the distribution. Of these, we can eliminate half by using the same memory area properly for several purposes. However, we must be careful in applying this overlay strategy, especially because the indexes in the original implementation are restricted to the range $1..n$. In addition, an array of $n/2$ integers is needed to store the list heads, an array of n integers to serve as cursors in the lists and an array of n records to store temporarily the keys to be sorted.

DDP is not directly comparable to *Dsort* in its original form: the $O(n^2)$ worst case occurs in *DDP* when all keys hit a single bucket. This happens when the distribution has a strong peak[5] or every tth element (t is the distance from one sample item to another) is the same. In addition, the method relies on the assumption of a predetermined sentinel element. Further, some complications occur in the determination of the coefficient for the mapping function when the keys are equal. Therefore we implemented two different versions: the first one, *DDP*, is gathered from the code fragments of ref. 13 and the second, *DDP_n* $\log n$, which

1. recognises the case when *samplemin* equals to *samplemax* and in this case uses quicksort for the whole array;
2. sorts large buckets with partial quicksort and the whole array with the insertion sort;
3. determines a sentinel element for the insertion sort from the *ins_level* elements of the last subfile.

(Note 3. If *samplemin* = *samplemax*, then we could also set $t = 1$ and resample the items. If after this *samplemin* still equals *samplemax* the sorting is done. Otherwise the processing is continued in a normal way.

Note 4. We could intertwine the grouping of the bucket elements and insertion sort in the original algorithm of Noga and Allison. This is 'safe', i.e. no large buckets are to be expected because of the double distribution. This was not implemented in *DDP_nlogn*, however, because then the algorithm would not be guaranteed to run in $O(n \log n)$ expected time any more.)

Complete program codes of *Dsort*, *Usort* and *DDP_nlogn* are available from the authors.

4. EXPERIMENTS FOR ARRAYS

The running time of distributive sorting algorithms depends on $k(=n/m)$, the average number of records in the buckets. The closer the bucket sizes are to the average size, the shorter is the time for sorting the buckets. Most of the buckets are sorted by insertion sort but large buckets which result in partitioning steps in *Quicks* become more probable as k increases. On the other hand, a small k (say $k = 1$) is not necessarily the best choice. Each bucket is to be initialised and empty buckets are checked at the bucket sort phase. Further, a complicating factor is the sensitivity of the expected running time on the distribution of the key values. To determine a reasonable bucket size we experimented with the various algorithms by using small k-values ($k = 1, 2, \dots, 10$) when sorting random keys from the interval $[0, 1)$. The best observed running times for *Dsort* and *Usort* were with $k = 5$ and for *DDP* and *DDP_nlogn* with $k = 2$. The results were not very sensitive to minor variations in k. In the tests to follow these k values have been used.

Table 1 shows the running times for *Dsort*, *Usort*, *DDP*, *Quicks* (= partial_quicks followed by insert _stopper_max), *DDP_nlogn* and *Linear probing sort*.[16] The last one runs in an $O(n)$ expected time for uniformly distributed points, has a very simple code and has been fast in computer runs with random keys (see ref. 8, table 4.5). From Table 1 is seen that the value of the two-level distribution becomes evident when the distribution is non-uniform. Still, excluding linear probing sort *Usort* outperforms all other methods even with the logarithmic distribution. Procedure *DDP* degenerates (given $O(n^2)$ running time), if the sample fails to give a reasonable approximation of the distribution function (see cases 7 and 8). In the same cases *DDP_nlogn* acts as a quicksort preceded by an unsuccessful distribution phase. *Linear probing sort* degenerates in several common cases.

Table 2 shows the duration of different phases of *Dsort*, *Usort*, *DDP* and *DDP_nlogn*. Here 1000 random keys in the range [0,1) were sorted. The first three rows show clearly the power of sampling: *DDP* and

Table 1. Comparison of *Dsort*, *Usort*, *DDP_nlogn*, *DDP*, *Linear probing sort* and *Quicks*

	Dsort	*Usort*	DDP_*nlogn*	*DDP*	*lprobing_sort*	*Quicks*
1. Random keys	5.81	5.01	6.86	6.87	4.01	7.04
2. Normal distribution	6.36	5.62	6.61	6.55	degen.	6.83
3. Logarithmic distribution	6.94	5.94	6.69	6.67	degen.	6.87
4. Equal keys	0.47	0.62	7.15	degen.	0.50	7.05
5. Increasing order	5.10	5.72	6.90	6.72	3.59	3.80
6. Decreasing order	6.25	4.17	6.23	6.22	3.51	4.50
7. Random, but each *k*th key is 0.5	6.52	5.90	6.92	degen.	degen.	6.81
8. Random, but each *k*th key is 0.5, the first key is 0.51	6.62	5.87	9.63	degen.	degen.	6.62
9. Random, with one exceptional key	11.40	10.50	6.77	6.77	degen.	6.85

Running times are in seconds for 16K records in MicroVAX-II. Results are averages from 25 test runs with distributions 1–3 and from 1 run in other cases. 'Degen.' denotes a stack space overflow or excessive time usage.

DDP_nlogn need on average only 5.9 s for sampling, determining the histogram and calculation of the mapping function whereas *Dsort* consumes 24.7 s for corresponding tasks (including a transfer of the records to another array). On the other hand the two former methods contain a more involved distribution phase which takes more than twice the time in *Dsort*. In addition *DDP* and *DDP_nlogn* perform linked list operations and the move of the result from a temporary array back to the original one (timings on rows 4 and 5).

Table 2. Components of the running time in sec, $n = 1000$ for Macintosh Plus Pascal, averages from ten runs. Random keys

	Dsort	*Usort*	*DDP*	*DDP_nlogn*
Determine min and max	8.0	7.6	2.1	2.1
Determine histogram	16.7	—	3.5	3.5
Calculate the mapping function	0.0	0.0	0.3	0.3
Distribute the records	18.3	18.4	38.2	38.2
Reorganise according to the linked lists and/or quicksort	4.7	12.6	9.2	11.3
Insertion sort	18.4	18.6	12.4	12.1
Dump the result to the original array	—	—	2.7	2.7
Total	66.1	57.2	68.4	70.2

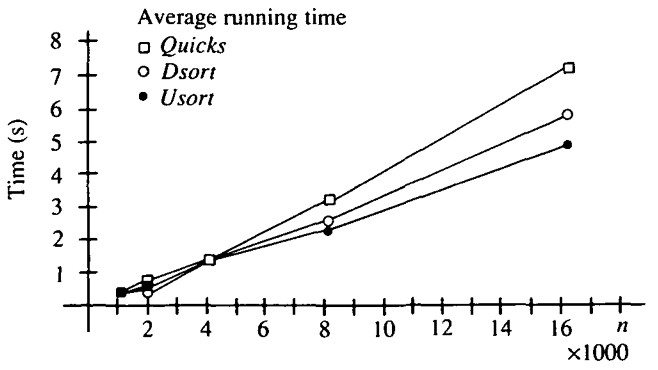

Figure 1. The observed running time for *Usort*, *Dsort* and *Quicks*. Uniformly distributed random points, MicroVAX-II. 25 repetitions.

The difference in the insertion sort times is due to the different average number of records in the buckets.

Fig. 1 shows the actual observed running times for *Dsort*, *Usort* and *Quicks*. The data sets constituted uniformly distributed random numbers. For *DDP* and *DDP_nlogn* the running times were slightly smaller than that of *Quicks* for $n > 8000$. For smaller values of n these procedures consumed more time than *Dsort* and *Quicks*. For files smaller than 2048 quicksort outperformed *Dsort*. Thus the *large_file* parameter value can be 2000 in our implementation (for the uniform distribution of the key values).

5. *DSORT* WITH LINKED LISTS

The top level logic of the stable linked list version of *Dsort*, named *Ldsort*, is closely related to that of *Dsort*, but we give it here to express the differences in the implementation of the substeps:

```
procedure ldsort(var ptr_to_first; nodepointer);
begin
  find_parameters(n,ptr_to_last,min,max));
  if number_of_items < large_file
  then
    partial_trisort(ptr_to_first,ptr_to_last,n)
  else
  if min < > max then
  begin
    distribute_records(ptr_to_first,min,max,m,bucket_
      heads,bucketrears,
        items_in_buckets);
    for i:= 1 to m do
      if items_in_buckets[i] > insertion_level
        then partial_trisort(bucketheads[i],bucketrears[i],
        items_in_buckets[i]);
    join_the_buckets(ptr_to_first,m,bucketheads,
      bucketrears);
  end;
  insertion_sort_using_sentinel(ptr_to_first);
end;
```

The *nodepointer* points to an entity which has the record type and a pointer of the *nodepointer* type as its components. All elements of the list from the node pointed by *ptr_to_first* to the one containing a nil-pointer in the link field are sorted.

The sort is initialised by determining the number of records, the pointer to the last node of the list, and the minimum and the maximum key values. The two former data are needed for the application of *partial_trisort* and the two latter ones to calculate the bucket boundaries.

If the number of records is small, then *Trisort*, a linked list variant[17] of *quicksort* is applied. *Trisort* is designed to handle efficiently key sets which contain identical keys: the records having a key value identical to the pivot are gathered together in the partition phase and excluded from the sequent recursive cells. The original procedure has been modified to avoid the stack space overflow by a standard technique where the sizes of the subfiles resulting from the partition guide the recursion. In addition, the recursion is stopped as soon as the subfiles become smaller than a predetermined threshold value *insertion_level*. Note that the best value of *ins_level* (9–10) in array based methods differs from the corresponding *insertion_level* (17–20) for linked lists. However, the actual value used does not affect significantly the running times due to the minor role of the *Trisort* in the distributive method. The median-of-three technique for choosing the pivot cannot be used because no direct access to the middle element of the list exists. The key of the last node is selected for the pivot[17] and thus, e.g. inputs with increasing or decreasing key sequences yield quadratic running times. The order of the items has been reversed in *partial_trisort* in order to apply the list based insertion sort flexibly.

For large arrays the processing is analogous to that in *Dsort* except that

- pointers to the rear (and front) of each bucket list are maintained,
- *partial_trisort* sorts the buckets in descending order, and
- the buckets are chained together starting from the one containing the largest key values.

The last two actions are needed for the application of the insertion sort.

(Note 5. The removal of the remaining disorder inside the buckets as the final step gives the possibility to relax the index calculation: it is not of primary importance to find the final bucket for each item in the distribution phase if it is guaranteed that they fall near it.)

Let us finally consider the stability of the method. The records are moved in the following operations:

(*a*) *Distribute_records.* The bucket lists are maintained by using front and rear pointers. The new item is always inserted to the rear of the list to preserve the stability.

(*b*) *Join_the_buckets.* Concatenation of the bucket lists is clearly stable.

(*c*) *Insertion_sort_using_sentinel.* Insertion sort is stable.

We implemented also a linked list version, *LDDP*, of the double distributive partition sort, *DDP*. Here the rewriting was done greatly in the same way as above.

6. PERFORMANCE OF THE LINKED LIST ALGORITHMS

The main advantage of the linked list representation over the array based one is that the copying of the records can be avoided. Its importance can be seen from the

Table 3. A comparison of the list and array versions of *Dsort* and *DDP*. Ten test runs with random data, MicroVax-II

Number of records to be sorted (K)	Average running time (s) of the algorithm					
	Ldsort	*LDDP*	*Trisort*	*Dsort*	*DDP*	*Quicks*
1	0.18	0.31	0.25	0.35	0.41	0.32
2	0.37	0.63	0.58	0.71	0.81	0.69
4	0.75	1.26	1.23	1.40	1.63	1.49
8	1.51	2.63	2.67	2.85	3.29	3.20
16	3.03	5.25	6.18	5.70	6.67	6.94

Table 3, which shows for uniformly distributed random keys the running times of *LDDP* and *Ldsort* and compares them with the corresponding array variants. List and array implementations of *Quicksort* are also included in the table.

Tests with normal and logarithmic distributions gave very similar results, showing that the one-level distribution of *Ldsort* is sufficient for 'smooth' distributions. Only in an exceptional case where one very large key value was included in the random key set, did the *Quicks* outperform the other methods.

Insertion sort works slower on linked lists than on arrays. Still, the difference is not very dramatic, if compared to the advantage gained when avoiding the moving of records between arrays. The role of the copying becomes even more critical when the record size grows.

The main reason for the weaker performance of *LDDP* as compared to *Ldsort* is the determination of the bucket boundaries which requires two traversals through the input list. This suggests that it is inappropriate to implement the original ideas of the double distributive partitioning sort with linked lists. A variety of possibilities for improving the performance could be devised, e.g. the sample could be collected into an array or into a sample list (a separate list which is extracted during the first scan). Furthermore, there is no reason to look for sample minimum and maximum because the whole list is scanned and the true values can be determined. This implies that the distribution becomes more straightforward. However, experiments showed that these changes do not improve the running time of *LDDP*: the work removed from the distribution is now done in the search of the extreme values.

For random keys *Ldsort* used only about half of the time of *Trisort*. In the original *Trisort* recursion is continued for subfiles until they contain equal keys. A modification where a threshold value is used to control the stopping of the recursion was implemented. As soon as the size of the subfile falls under the threshold, it is left unsorted. The insertion sort is applied as a final step to take care of the possible disorder which is still left. This modification did not, however, improve the times of the original *Trisort* noticeably (the improvement was of the order of 4%). Also an attempt to make *Ldsort* more elegant by omitting the insertion sort and *Trisort*ing all buckets resulted in a weaker performance.

When comparing the running times of *Quicks* and *Trisort*, one must bear in mind that the latter does not contain a sophisticated pivot selection (e.g. the median-of-three).

The space requirements of the algorithms are dependent on the environment where the sort routine is applied: if the input file is given in an array, the space demand does not decrease from that of *Dsort* (or *DDP*), i.e. the list space created exceeds the space needed for an auxiliary array. (In addition one still has to pay for the time of constructing the linked list so that the application of the method is hardly profitable in these cases.) However, there is frequently an option to give the input in either form, for example when the records are extracted from a larger data file as a preprocessing step. The linked representation manages without an extra space for a copy of the records. Thus the space savings may be considerable, although space for n additional links plus $2m$ additional pointers for maintaining the buckets during the distribution process must be reserved.

5. CONCLUDING REMARKS

Dsort and *Ldsort* are general-purpose sorting algorithms which are based on the distributive sorting technique. For smooth distributions of the key values their expected running time depends linearly on the number of the records to be sorted. The procedures were compared to three other bucket sort algorithms along with efficient implementations of quicksort.

Dsort has the following characteristics:

(*a*) It needs less memory than the other array based bucket sort algorithms tested in this paper. All these (including *Usort*) need an extra array to hold the n records temporarily. This feature makes them less attractive than the 'conventional' comparison based methods, if space is a critical resource.

(*b*) It runs very fast for smooth distributions. When the distribution has a strong peak, the strength of the double distribution of *DDP* becomes evident.

All in all, *Dsort* is a 'compromise': it needs more space than quicksort but less than other bucket sort methods. On the other hand, it runs faster than quicksort but slower than, at least, *Usort*.

The results for *Ldsort* are still more encouraging. It was noticed that if we do not have any constraints on the choice of the data structure, the linked list should be preferred to an array representation. There exist several facts in favour of a linked list:

(*a*) it has the basic property of a dynamic structure, i.e. it can easily be extended or shrunk to adapt to the input size,
(*b*) the sorting is stable,
(*c*) the space demand of main storage is small, and
(*d*) the running times are significantly smaller.

If the size of the original record is large, the input is usually given as a *tag file* where each item to be sorted contains only the key value and the relative position of the record in the original file reducing thereby the storage demand of the array based methods somewhat. Nevertheless, even this does not make them comparable to the linked alternative. An important situation where the linked list is not appropriate, is the one where the result of the sorting is used in a way which assumes direct access to the sorted sequence.

As to the practical usefulness of the bucket sort, one must keep in mind that

(*a*) the fields according to which the sorting is done, must be numeric or, if this is not the case, we must have an order-preserving mapping from the key values to (a subset of) real values of finite range;
(*b*) overflow and underflow are possible in the arithmetic operations of the procedure;
(*c*) the worst case is even weaker than that of quicksort due to the preliminary steps before the quicksort is applied, and
(*d*) the demand for the extra storage is relatively large.

After the first writing of the paper we have implemented another version of *Dsort* where the extremal key values are only approximated from a sample of $n/100$ elements. This is a fast process for arrays. In addition, exceptionally large and small keys will be bypassed with high probability. However, the fact that *samplemin* equals to *samplemax* does not imply that all keys are identical, and thus we have to continue by performing quicksort. As a net effect, the running times for distributions 1,2,3,5,6 (Table 1) decreased by approximately 14% whereas they increased for distributions 8,9 about 25%.

REFERENCES

1. S. G. Akl and H. Meijer, Hybrid sorting algorithms. *Proceedings of the 18th Annual Allerton Conference on Communication, Control and Computing*, pp. 250–259 (1980).
2. S. G. Akl and H. Meijer, On the average-case complexity of 'bucketing' algorithms. *Proceedings of the Annual Allerton Conference on Communication and Computing*, pp. 381–388 (Oct. 1981).
3. D. C. S. Allison and M. T. Noga, USORT: an efficient hybrid of distributive partitioning sorting. *BIT* **22**, 135–139 (1982).
4. W. Dobosiewicz, Sorting by distributive partitioning. *Inf. Proc. Lett.* **8**, 4, 168–169 (1979).
5. L. Devroye and T. Klincsek, Average time behaviour of distributive sorting algorithms. *Computing* **26**, 1–7 (1981).
6. L. Devroye, Lecture notes on bucket algorithms. In *Progress in Computer Science*, vol. 6. Birkhäuser (1986).
7. J. Ernvall, O. Nevalainen and T. Raita, Performance tests with distributive sorting programs. *Proceedings of the 1982 DECUS Europe Symposium, Warwick, UK*, pp. 51–57.
8. G. H. Gonnet, *Handbook of Algorithms and Data Structures.* Addison-Wesley (1984).
9. J. S. Kowalik and Y. B. Yao, Implementing a distributive sorting program. *Journal of Information and Optimisation Sciences* **2**, 1, 28–33 (1981).
10. H. Meijer and S. G. Akl, The design and analysis of a new hybrid sorting algorithm. *Inf. Proc. Letters*, **10**, 4–5, 213–218 (1980).
11. M. van der Nat, A fast sorting algorithm, a hybrid of distributive and merge sorting. *Inf. Proc. Letters* **10**, 3, 163–167 (1980).
12. O. Nevalainen and J. Ernvall, Implementation of a distributive sorting algorithm. Technology and Science of Informatics **2**, 1 (1983).
13. M. T. Noga and D. C. S. Allison, Sorting in Linear expected time *BIT* **25**, 451–465 (1985).
14. M. T. Noga, Fast geometric algorithms, Ph.D. Dissertation, Virginia Polytechnic Institute and State University (1984).
15. R. Sedgewick, Implementing quicksort programs. *Comm. of the ACM* **21**, 10, 847–857 (1978); see also Corrigendum, *Comm. of the ACM* **22**, 6, 368 (1979).
16. J. Teuhola, private communication, 1991.
17. L. M. Wegner, Sorting a linked list with equal keys. *Inf. Prog. Lett.* **15**, 5, 205–208 (1982).