Three New Sorting Algorithms Based on a Distribute-Collect Paradigm

P. Y. Padhye

(puru@deakin.edu.au)

School of Computing and Mathematics
Deakin University
Geelong, Victoria 3217

Abstract

Three new sorting algorithms, called StackSort, DeqSort and MinMaxSort, are described. They are of interest for the following reasons: they are adaptive sorting algorithms; they are comparison based general sorting algorithms; they do not put any restriction on the type of keys; they use linked lists; the moves or exchanges of data required in algorithms using arrays are unnecessary; the desired order is obtained by adjustment of the pointers; they provide examples of interesting applications of data structures such as stacks, queues, and singly- and doubly-linked lists.

A new and improved variation of the well known Natural MergeSort, called here SublistMergeSort, is also presented.

These algorithms are compared with one another and with other well known sorting algorithms such as InsertionSort and HeapSort in terms of times required to sort various input lists. Input lists of various sizes and degrees of 'presortedness' were used in the comparison tests. It has been demonstrated that the performance of DeqSort and MinMaxSort is better than InsertionSort and comparable to HeapSort; and that StackSort performance is better than InsertionSort in most cases.

1. Introduction

Sorting is the most-studied and most-solved problem in the theory of Algorithms [Kingston, 1990]. Sorting and searching are among the most frequently occurring activities in computing. I.B.M. estimates that about 25% of total computing time in commercial computing centers is spent on sorting alone [Mehlhorn,1984]. The topic has been researched for many years and several books are written on this topic [Knuth, 1973], [Mehlhorn, 1984], [Sedgewick, 1980]. Sorting is an almost universally performed, fundamental, relevant and essential activity, particularly in data processing [Wirth, 1976]. Sorting and searching are also important topics in the area of data structures. Hence research in this area is still lively and relevant.

In this report three sorting algorithms which are based on a two phase paradigm, called here a Distibute-Collect paradigm, are described.

In the first phase, called the distribution phase, the items in the input list are distributed into sublists of ordered subsequences. The sublists are similar to the runs identified in the Natural MergeSort algorithm. However, unlike the runs in Natural MergeSort, the items in each sublist are not necessarily adjacent in the input. The distribution phase involves only one pass through the input list.

In the second phase, called here the collection phase, items are collected progressively from the sublists to build the sorted output list. The collection phase may involve several repetitions.

Some examples of sorting algorithms based on the two phase paradigm are MergeSort, Natural MergeSort, RadixSort [Aho, 1974], [Kingston, 1990].

Radix sort is suitable only for integer data. The algorithms described here are suitable for integer as well as character data.

The DeqSort and MinMaxSort are better than Natural MergeSort as they identify runs which are longer than the runs identified by Natural MergeSort.

The StackSort, DecSort and MinMaxSort algorithms are based only on comparison of input data elements. They involve no data exchanges, which are required in most other algorithms such as Insertion Sort, Selection Sort, Heap Sort and Quick Sort. When the size of the elements to be sorted is large, absence of data exchanges is an advantage.

The StackSort, DeqSort and MinMaxSort algorithms are adaptive. Moffat and Petersson describe an algorithm as adaptive if 'easy' problem instances are solved faster than 'hard' instances.[Moffat,1992]. StackSort can sort 'easy' problem instances faster than 'hard' instances. DeqSort and MinMaxSort do better than StackSort. They can sort 'hard' problem instances almost as fast as 'easy' instances.

Most internal sorting algorithms need the input data to be read into an array for sorting. StackSort, DeqSort and MinMaxSort do not need to do this. The data read from an input file can be inserted straight into an appropriate sublist in the distribution phase.

The following Sections 2, 3, 4 and 5 describe the sorting algorithms. For the purpose of illustrating these algorithms it is assumed that sorting is required in ascending order (non decreasing order) with duplicates allowed.

2. StackSort

This algorithm is called StackSort here as it uses the data structure stack in
both the distribution and collection phases, which are described below.

2.1 The distribution phase of StackSort

The distribution phase uses the algorithm described by Moffat and Petersson [Moffat, 1992].
In this phase the input list is distributed into sublists in such a way that:

- each sublist contains an ascending subsequence from head to tail, the tail item is greater than or equal to every other item in the sublist.

- the sublists are also ordered so that their tail items are in descending order. The tail item of the first sublist is greater than the tail item of the second sublist, which is greater than the tail item of the next sublist and so on.

This algorithm works as follows.

The first item of the input starts the first sublist, becoming its first member. Each of the remaining items in the input list is considered in turn. It is compared with the tail items of the existing sublists and, if possible, appended to the first sublist that has its tail item smaller than or equal to the input item. If all the existing sublists have tail items greater than the input item, a new sublist is started with the input item as its first member.

For example, consider the following input list.

(9, 5, 6, 3, 8, 7, 1, 11, 2, 10, 5, 12)

The first item, 9, will start sublist1. The second item, 5, being smaller than 9, will start sublist2. The next item, 6, being greater than 5, will be appended to sublist2. Next item 3 will start sublist3. Next item 8 will be appended to sublist2, and so on. Continuing in this fashion, the input list will be distributed into the following sublists when all the items are read.

sublist1	9	11	12	
sublist2	5	6	8	10
sublist3	3	7		
sublist4	1	2	5	

Note that at the end of the distribution phase:

- The items in each sublist are in increasing order from head to tail constituting one 'run', but the items are not adjacent in the input.
- The tail items of the sublists are in descending order. The tail item of sublist1 is the largest item in the input list.
- The number of sublists, 4, is less than the number of runs, 7, in the original data.

2.2 The collection phase of StackSort

The tail item of sublist1, the maximum item, is deleted from that list and written to the output list. The sublists are rearranged, if necessary, such that the tail items continue to be in descending order. The tail item of sublist1 continues to supply the next maximum to be written to the output list. The process continues until all items are written to the output list.

The collection phase of StackSort is described by the pseudo-code shown in Fig.1.

```
WHILE (no_of_sublists > 1) do

    REPEAT
        from the tail end of the first sublist extract items greater than or equal
        to the tail item of the second sublist and write these to the output list
        (which is collecting the items in descending order)

    UNTIL (first_sublist_tail_item < second_sublist_tail_item)

    if the first sublist is not empty, insert it in the group in an appropriate
    place such that the tail items of the sublists are in descending order again

END_WHILE

When only one sublist is left write its items, from tail to head, to the output list
```

Fig.1. Pseudo-code for the collection phase of StackSort

Since all the actions on each sublist take place on its tail item, the stack structure is ideal for the sublist implementation. The output list also collects the items in reverse, descending, order. The output list may also be implemented as a stack.

Working of the collection phase may be illustrated using the sublists above. The sublists are now designated as stacks S1, S2, etc. The following stacks were formed at the end of the distribution phase:

```
S1  9  11  12
S2  5   6   8  10
S3  3   7
S4  1   2   5
```

(Stacks are shown growing to the right; the right most element is at the top.)

Start by popping 12 from S1 and pushing it on to the output stack. This is followed by popping 11 from S1 and pushing it on to output stack. Now the stack S1 has its tail item, 9, less than the tail item of the second stack. Hence the stacks are reordered as follows to maintain the tail items in descending order.

```
S2  5   6   8  10
S1  9
S3  3   7
S4  1   2   5
```

Now 10 is popped from the first stack S2, and the stacks reordered:

```
S1  9
S2  5   6   8
S3  3   7
S4  1   2   5
```

Continuing in this fashion:

```
pop 9 :  S2  5   6   8      pop 8 :  S3  3   7          pop 7 :  S2  5   6
         S3  3   7                   S2  5   6                   S4  1   2   5
         S4  1   2   5               S4  1   2   5               S3  3

pop 6 :  S2  5              pop 5 :  S4  1   2   5      pop 5 :  S3  3
         S4  1   2   5               S3  3                       S4  1   2
         S3  3

pop 3 :  S4  1   2
```

Now the single stack S4 will be popped out.

2.3 Comments on StackSort

The group of stacks is managed by maintaining an ordered linked list of pointers to the stacks. The individual stacks are also maintained as linked lists. Arrays are unsuitable for the stacks and also for the pointers to them as the size of the individual stacks and the pointers list can vary widely between 1 and N, where N is the size of the input.

The algorithm is sensitive to the order already existing in the input:

- The best case occurs when the input is already in increasing order, because only one sublist holding all the N items would be formed.

- The worst case occurs when the input is already sorted in reverse order, because there will be N sublists, each containing only one item.

The algorithm involves only comparisons between items and no data exchanges.

The algorithm selects maximum items like HeapSort. The maximum item is the tail item of the first sublist. There is little work involved in the maintenance of the sublists in the required order.

However, on the negative side, there is the overhead of maintaining pointers for the linked lists and extra space requirement.

3. DeqSort

The main disadvantage of StackSort is its worst case behaviour when the input list is sorted in the reverse order. This type of input can be tackled as efficiently as in the case of input in the right order if the sublists are implemented as double ended queues (Deqs): an item greater than the tail item of a sublist is appended to it, an item less than the head item of a sublist is prepended to it. This constitutes the basic difference between StackSort and DeqSort.

3.1 The distribution phase of DeqSort

The first item of the input starts the first sublist, becoming its first member. Each of the remaining items in the input list is considered in turn. It is compared with the tail item of the first sublist and if it is greater than or equal to the tail, the input item is appended to the list. Otherwise it is compared with the head item, and prepended to the list if it is smaller than or equal to the head item. If it cannot be appended or prepended to the sublist, similar comparison continues with the second, or subsequent sublists, and the item is appended or prepended to the first suitable sublist. If the item cannot be appended or prepended to any existing sublist, a new sublist is started with the input item as its first member.

Each sublist holds one subsequence in ascending order from head to tail. The tail item in each sublist is the largest in that sublist. The sublists themselves are also ordered such that their respective tail items are in descending order. In addition, the head items are in ascending order.

For example, consider the same input list which was used to illustrate the StackSort in Section 2.1.

(9, 5, 6, 3, 8, 7, 1, 11, 2, 10, 5, 12)

The sublists SL1, SL2,....will be built as follows.

```
read 9 :  SL1   9            read 5 :  SL1  5   9        read 6 :  SL1   5   9
                                                                   SL2   6

read 3 :  SL1   3   5   9    read 8 :  SL1  3   5   9    read 7 :  SL1   3   5   9
          SL2   6                      SL2  6   8                  SL2   6   8
                                                                   SL3   7
..... and so on.
```

At the end this phase the following sublists will be produced:

```
SL1  1   3   5   9  11  12
SL2  2   6   8  10
SL3  5   7
```

Note that

- The number of sublists, 3, is smaller than the number of runs, 7; and also smaller than the number of sublists produced by StackSort.
- The head items are in ascending order.

3.2 The collection phase of DeqSort

In the collection phase one can extract either the head items (current minimum) or the tail items (current maximum) from the first sublist. The description below is for the option of extracting head items. The extraction of head items results in writing to output in ascending order. Once an item is extracted, it will not be simple to reorder the sublists so as to maintain both the head and the tail items in the required order. Thus only the head items will be maintained in ascending order.

The collection phase of the DeqSort algorithm is described by the pseudo-code shown in Fig.2.

```
WHILE (no_of_sublists > 1) do

    REPEAT
        • extract from the head side of the first sublist items less than or
          equal to the head item of the second sublist
        • write them to the output list
          (which is collecting the items in the required ascending order)

    UNTIL (first_sublist_head_item > second_sublist_head_item)

    IF  the first sublist is not empty insert the old first sublist in an
        appropriate position so that the head items are again in ascending order

END_WHILE

When only one sublist is left
    write its items, from head to tail, to the output list
```

Fig.2. Pseudo-code for the collection phase of DeqSort algorithm

Working of the collection phase may be illustrated using the following sublists which were formed at the end of the distribution phase:

```
SL1  1   3   5    9  11  12
SL2  2   6   8  10
SL3  5   7
```

Start by dequeueing 1 from SL1 and writing it to the output list. Since the new head of SL1 (3) is greater than head of SL2, reorder the lists:

```
SL2  2   6    8  10
SL1  3   5    9  11  12
SL3  5   7
```

Dequeue 2, append it to output list, reorder the lists:

```
SL1  3   5   9   11  12
SL3  5   7
SL2  6   8  10
```

Continuing in this fashion:

```
Deque 3 :  SL1   5   9   11   12     Deque 5 :  SL3   5   7
           SL3   5   7                          SL2   6   8   10
            SL2  6   8   10                     SL1   9   1   12


Deque 5 :  SL2   6   8   10          Deque 6 :  SL3   7              Deque 7 :  SL2   8   10
           SL3   7                              SL2   8  10                     SL1   9   11  12
           SL1   9   1   12                     SL1   9  11   12

Deque 8 :  SL1   9   1   12          Deque 9 :  SL2  10              Deque 10 :  SL1 11   12
           SL2  10                              SL1  11  12          SL1 can be written out.
```

3.3 Comments on DeqSort

The group of sublists is managed by maintaining an ordered linked list of pointers to the head and tail of each sublist.

Each individual sublist is also implemented as a linked list. Arrays are unsuitable for the sublists and also for the pointers to them as mentioned earlier in the case of StackSort. However, the output list may be implemented as an array.

DeqSort has the potential to reduce the number of sublists considerably.

With DeqSort the best case occurs when the input is already in order, whether in right or reverse order does not matter. In both cases only one sublist holding all the N items would be formed. The extreme worst case of N sublists, each containing only one item which could occur in StackSort, will never occur in DeqSort.

This algorithm involves only comparisons between items and no data exchanges. The selection of current items (maximum or minimum) is like in HeapSort. There is work involved in the maintenance of the sublists in the required order. The data exchanges required in HeapSort to maintain the heap are not required in this algorithm.

However, as in the StackSort algorithm, there is the overhead of maintaining pointers for the linked lists and extra space requirement.

4. MinMaxSort

At the end of the distribution phase of DeqSort (Section 3), the sublists are ordered in such a way that the head items are in ascending order and the tail items are in descending order. The collection phase however maintains only the head items in ascending order while extracting the minimum items. MinMaxSort is a variation of DeqSort in which both the minimum and the maximum items are extracted at the same time. This would require that the sublists maintain both the head and the tail items in the required order.

4.1 The distribution phase of MinMaxSort

This is identical to the distribution phase of DeqSort and is already described in Section 3.1.

4.2 The collection phase of MinMaxSort

The MinMaxSort extracts items from both ends of the first sublist as follows.

From the head side of the first sublist it retrieves all items smaller than or equal to the head item of the next sublist. From the tail side of the first sublist it retrieves all items greater than or equal to the tail item of the next sublist. Items extracted from the head side are written to the lower end of output list, the ones extracted from the tail end are written to higher end of output list.

The rest of the items from the first list, if any, are inserted in the correct order between the head and tail of the second sublist. The number of sublists is thus reduced by one: the second being now the first, and the head and tail items of the sublists continue to be in the required order.

Extraction of head/tail items and reinsertion of middle items continues in this way until only one sublist is left. The last one is then written to the output list.

The pseudo-code shown in Fig. 3 explains the collection phase of MinMaxSort.

```
REPEAT
    IF there are 2 or more sublists left

        extract the head end items from the first sublist, which are smaller than or
        equal to the head item of the second sublist

        write them to the head end of output list

        extract the tail end items from the first sublist, which are greater than or
        equal to the tail item of the second sublist

        write them to the tail end of output list

        IF the first sublist is not empty

            insert the rest of first sublist in the second sublist in appropriate order.
            Note that the rest of the items are in increasing order, the second sublist
            is also in increasing order, conditions which are ideal for the
            insert operation.

        END_IF

    END_IF

UNTIL only one sublist is left

When only one sublist is left write its items, from head to tail, to the head end of
output list
```

Fig.3. Pseudo-code for the collection phase of MinMaxSort

Working of the collection phase may be illustrated using the sublists produced in Section 3.1.

```
SL1  1   3   5   9  11  12
SL2  2   6   8  10
SL3  5   7
```

Operate on the first sublist SL1.

- Extract 1 from head end, write it to left section of output, extract 12 and 11 from tail end, write them to right section of output, insert the rest (3 5 9) in the second list SL2.

This results in the following sublists:

```
SL2   2   3   5   6   8   9  10
SL3   5   7
```

Operate on SL2.

- Extract 2, 3, 5 from head end, write to left section of output, and extract 10, 9, 8 from tail end, write to right section of output, insert the rest (6) in SL3.

Now a single sublist is left. SL3 5 6 7.

SL3 is written to the output list.

4.3 Comments on MinMaxSort

MinMaxSort, similar to DeqSort, handles input ordered in any way almost equally well. The best case occurs when input is already sorted, whether right or reverse order does not matter.
To facilitate the append/prepend operations in the distribution phase and also the extraction of the head and tail items in the collection phase, the sublists are best maintained as doubly linked lists.

MinMaxSort has the potential to reduce the number of sublists significantly. However, it is more complicated than DeqSort and has higher storage overhead as the sublists are doubly-linked.

Two variations of MinMaxSort algorithm have been implemented in the tests described in Section 6.

- MinMax1: The output list is implemented as an array. Extra storage space is required for the array, but the the code is simpler.
- MinMax2: The output list is implemented as a doubly linked list. Sections from the left part of the top sublist are appended to the left part of output list, and sections from the right part of the top sublist are prepended to the right part of the output list. Since the append/prepend operations only readjust the pointers of already allocated nodes, the output list does not need extra storage. A contiguous subsection of the list can be handled more efficiently as an entity instead of handling each element separately.

5. SubListMergeSort (SLMerge Sort)

The distribution phase of DeqSort (or MinMaxSort) identifies longer ordered subsequences, and results in reduced number of 'runs' as compared to natural MergeSort.

Natural MergeSort may be improved by replacing its first phase of identifying the runs by the distribution phase of DeqSort (Section 3.1). The second phase would then simply merge the sublists. This combination of the distribution phase of DeqSort and the second phase of natural MergeSort is referred to as SublistMergeSort (abbreviated to SLMergeSort) here.

In SLMergeSort the distribution phase is identical to the first distribution phase used in DeqSort. The second phase merges the sublists two at a time until only one list is left.

The second phase is illustrated below for the same input list used in all the examples above. At the end of the distribution phase we have the following sublists.

```
SL1  1   3   5   9  11  12
SL2  2   6   8  10
SL3  5   7
```

Merging SL1 and SL2 into one list, SL12, we have

```
SL12 1  2  3  5  6  8  9  10  11  12
SL3  5  7
```

Merging SL12 and SL3 we have the final list, SL123:

```
SL123   1  2  3  5  5  6  7  8  9  10  11  12
```

The number of natural runs in the input data above are 7, but the runs formed in the SLMergeSort are only 3.

6. Comparison of the sorting algorithms

The above algorithms were tested on several input lists with varying degrees of presortedness. The lists contained integers in ascending, descending or various other configurations.

To supply the test lists the following files containing integers were generated.

- rndnos.4K 4096 random integers were generated using the Sun Pascal function random( ) on eros, a SUN-4 minicomputer, and written to this file.
- ascdnos.4K 4096 integers from rndnos.4K file were sorted in ascending order and written to this file.
- descdnos.4K 4096 integers from rndnos.4K file were sorted in descending order and written to this file.
- incnos.4K contained 4096 random integers in roughly increasing order. The integers (n) were generated using equation 1:

  $$n = 1024*k + r \qquad (1)$$
  for $k = 1,2,3,...,4096$
  where r is a random integer in the range 0 to 16383 [Hale,1992].

- decnos.4K contained 4096 random integers in roughly decreasing order. The integers (n) were generated using equation 2:

  $$n = 1024*(4097-k) + r \qquad (2)$$
  for $k = 1,2,3,...,4096$
  with r as above [Hale, 1992].

6.1 First set of tests with input lists of 1024 integers

Since the sorting tests were run on a relatively slow IBM PC compatible, the input lists used a smaller subset, the first 1024 numbers, contained in the files described in Section 6.

The input lists were sorted using StackSort, DeqSort, Minmax1, Minmax2, and SLMergeSort algorithms. The InsertionSort and HeapSort algorithms were also used for comparison of the sorting times required.

The times required for sorting were obtained on an IBM PC compatible using Turbo Pascal programs.

The times required for sorting are given in seconds in Table 1. They may be considered to be adequate for comparison purposes only.

Table 1. Comparison of Times (Seconds) for Sorting 1024 Integers

Inputfile	<--------		Sorting Method Used				------->
	Insertion	StackSort	DeqSort	MinMax1	Minmax2	HeapSort	SLMerge
ascdnos.4K	0.94	1.07	1.12	1.26	1.26	1.53	1.21
incnos.4K	0.99	1.07	1.05	1.10	1.10	1.18	0.99
rndnos.4K	7.70	1.90	1.95	1.78	1.75	1.20	1.81
decnos.4K	14.61	7.06	1.11	1.12	1.15	1.12	1.10
descdnos.4	14.78	14.15	0.94	0.99	0.98	1.12	0.93

Table 1 shows that DeqSort, MinMax1, MinMax2, SLMergeSort are comparable to HeapSort. DeqSort, MinMax1, MinMax2 and SLMergeSort perform somewhat better when the input is already ordered, whether ascending or descending does not matter. But they handle even random input much faster than the Insertion sort.

The StackSort performance is comparable to Insertion sort, handling input data data already in the correct order very well, while performing poorly in the case of data in reverse order. However, it handles random data much better than Insertion sort and almost as well as DeqSort, MinMax1 and MinMax2.

There is no appreciable advantage in choosing the more complicated MinMax1 or MinMax2 over the simpler DeqSort.

6.2 Second set of tests on input lists of various sizes

Encouraged by the results of the first set of tests described in Section 6.1 further tests were carried out using the files described in Section 6 with the size of input list varying from 64 to 4096. The timing information is again based on running Turbo pascal programs on an IBM PC compatible.

The results of these tests are shown in Tables 2 to 6.

Table 2. Comparison of Sorting Algorithms (Input file ascdnos.4K)

Algorithm	Sorting Time (seconds) — No. of integers sorted in ascending order						
	64	128	256	512	1024	2048	4096
Insertion Sort	0.03	0.07	0.13	0.33	0.64	1.14	2.10
StackSort	0.04	0.11	0.21	0.52	0.99	1.90	3.65
DeqSort	0.05	0.12	0.24	0.54	1.05	1.96	3.77
MinMax1	0.04	0.11	0.23	0.53	1.01	1.94	3.76
MinMax2	0.04	0.10	0.22	0.55	1.03	1.87	3.61
SLMergeSort	0.04	0.11	0.21	0.51	0.99	1.86	3.66
HeapSort	0.06	0.13	0.26	0.61	1.23	2.42	4.95

The results shown in Table 2 confirm that simple insertion sort is the best algorithm when the data is already in the required order. All the other algorithms have comparable times. The algorithms StackSort, DeqSort, Minmax1, MinMax2 and SLMergeSort all seem to be a little better than HeapSort for this case.

Table 3. Comparison of Sorting Algorithms (Input file incnos.4K)

Algorithm	Sorting Time (seconds) — No. of integers sorted in ascending order						
	64	128	256	512	1024	2048	4096
Insertion Sort	0.04	0.10	0.23	0.42	0.87	1.80	3.51
StackSort	0.04	0.14	0.28	0.53	1.02	2.08	4.12
DeqSort	0.06	0.15	0.26	0.51	1.04	2.09	4.14
MinMax1	0.06	0.12	0.28	0.49	0.97	1.98	3.94
MinMax2	0.06	0.13	0.26	0.49	0.97	1.97	3.89
SLMergeSort	0.04	0.12	0.24	0.47	0.98	2.00	3.94
HeapSort	0.06	0.12	0.25	0.52	1.06	2.28	4.77

Table 3 shows that when data is in roughly increasing order, simple Insertion sort is best, but the other algorithms also have comparable times.

Table 4. Comparison of Sorting Algorithms (Input file rndnos.4K)

Algorithm	Sorting Time (seconds)						
	No. of integers sorted in ascending order						
	64	128	256	512	1024	2048	4096
Insertion Sort	0.06	0.20	0.60	2.13	7.62	29.65	115.61
StackSort	0.07	0.14	0.32	0.78	1.87	4.75	12.32
DeqSort	0.06	0.15	0.31	0.71	1.74	4.37	11.03
MinMax1	0.06	0.16	0.31	0.68	1.66	4.07	10.21
MinMax2	0.05	0.14	0.31	0.67	1.65	4.04	10.13
SLMergeSort	0.06	0.16	0.30	0.70	1.79	4.68	12.05
HeapSort	0.04	0.12	0.24	0.49	1.05	2.25	4.76

Table 4 indicates that when data is in random order, StackSort outperforms Insertion Sort. DeqSort, MinMax1, MinMax2, SLMergeSort and HeapSort are all comparable with one another and are all better than StackSort. HeapSort is better than DeqSort, MinMax1, MinMax2 and SLMergeSort when the input size is very large (>1024). For small input size (< 128) Insertion sort performs as well as others.

Table 5. Comparison of Sorting Algorithms (Input file decnos.4K)

Algorithm	Sorting Time (seconds)						
	No. of integers sorted in ascending order						
	64	128	256	512	1024	2048	4096
Insertion Sort	0.09	0.31	1.03	3.76	14.39	56.39	222.77
StackSort	0.07	0.21	0.61	1.97	6.65	25.71	98.79
DeqSort	0.05	0.14	0.25	0.55	1.01	2.10	4.14
MinMax1	0.04	0.13	0.25	0.50	0.99	2.03	3.96
MinMax2	0.05	0.12	0.25	0.49	0.96	1.99	3.56
SLMergeSort	0.04	0.14	0.24	0.48	0.97	2.02	3.99
HeapSort	0.06	0.12	0.24	0.48	1.02	2.21	4.57

Table 5 shows that when data is in roughly decreasing order, Insertion sort performs poorly. StackSort is better than Insertion sort, but not as good as the other algorithms. DeqSort, MinMax1, MinMax2, SLMergeSort and HeapSort have all comparable performance much better than Insertion sort.

Table 6. Comparison of Sorting Algorithms (Input file descdnos.4K)

	Sorting Time (seconds)						
Algorithm	No. of integers sorted in ascending order						
	64	128	256	512	1024	2048	4096
Insertion Sort	0.08	0.31	1.08	3.88	14.72	57.41	227.07
StackSort	0.09	0.31	1.08	3.79	14.13	55.01	218.02
DeqSort	0.04	0.11	0.27	0.48	0.94	1.84	3.67
MinMax1	0.03	0.12	0.25	0.47	0.92	1.82	3.57
MinMax2	0.03	0.11	0.25	0.46	0.91	1.77	3.52
SLMergeSort	0.04	0.10	0.24	0.47	0.90	1.75	3.47
HeapSort	0.06	0.12	0.25	0.52	1.05	2.18	4.59

Table 6 demonstrates that when data is in reverse order the results are similar to the case of data in roughly decreasing order. Insertion sort and StackSort performance is poor, except when the input size is very small. Performance of DeqSort, MinMax1, MinMax2, SLMergeSort is good and comparable to the case of input already sorted in the required order. These algorithms seem to be somewhat better than HeapSort in this case.

7. Comparison of runs produced in DeqSort with natural runs in the lists

As compared to Natural Merge Sort, DeqSort identifies longer runs. This results in reduced number of initial sublists produced by DeqSort as compared to Natural Merge Sort. Table 7 shows this comparison for the input lists used in the tests described in Section 6.1.

Table 7. Comparison of the Number of Sublists Produced

	Number of Sublists Produced Input first 1024 Integers from each file		
Input File	Natural MergeSort	DeqSort	Input List property
ascdnos.4K	1	1	ascending order
incnos.4K	441	6	roughly increasing order
rndnos.4K	507	39	random order
decnos.4K	588	6	roughly decreasing order
descdnos.4K	892	1	descending order

Table 7 shows that DeqSort produces substantially smaller number of sublists than Natural MergeSort.

The sorting time is reduced when the number of initial sublists is reduced. Hence the SLMergeSort algorithm which is identical to DeqSort so far as the distribution phase producing the sublists is concerned, would be somewhat faster than Natural MergeSort. Note also that the distribution phase of MinMaxSort is identical to the one of DeqSort. Hence MinMaxSort would also produce smaller number of sublists than Natural MergeSort.

8. Conclusions

Three new sorting algorithms, StackSort, DeqSort, MinMaxSort, and an improved version of natural MergeSort, called SublistMergeSort (SLMergeSort in short) are described. These algorithms belong to the class of the algorithms based on a two phase paradigm called a distribute-collect paradigm here. They are compared with one another and with InsertionSort and HeapSort in terms of the times required to sort inputs of various degree of 'presortedness'.

The results presented in Section 6.2 confirm the known efficiency of simple Insertion sort for very small input sizes and also for input already in the required order. It can be seen that DeqSort, MinMax1, MinMax2, SLMergeSort are comparable to HeapSort in most cases. DeqSort, MinMax1, MinMax2 and SLMergeSort perform somewhat better when the input is already ordered, whether ascending or descending does not matter. But they handle even random input much faster than Insertion sort when the input size is not small.

The StackSort performance is comparable to Insertion sort, handling input data already in the correct order very well, while performing poorly in the case of data in reverse order. However, it handles random data much better than Insertion sort and almost as well as DeqSort, MinMax1 and MinMax2.

There is no appreciable advantage in choosing the more complicated MinMax1 or MinMax2 over the simpler DeqSort.

The distribution phase used by DeqSort, MinMaxSort and SLMergeSort is more effective than the one used in Natural MergeSort in producing much smaller number of ordered sublists.

Like HeapSort DeqSort, MinMaxSort and SLMergeSort can handle input in any order equally well. Unlike HeapSort they do not involve data exchanges. This is a factor in their favour when the size of elements to be sorted is large.

Acknowlegements

The author would like to thank Prof. Andrzej Goscinski and Mr.Bob Hale for their encouragement, criticism and useful comments.

Thanks also to Judith Jamieson for all the secretarial help.

References

Aho, A.V., Hopcroft, J.E., Ullman, J. D., 1974.
"The Design and Analysis of Computer Algorithms"
Addison-Wesley.

Hale, R.P., Teague, G.J., 1992.
"SCP760 Data Structures Study Guide",
Session 8, p. 14.
Deakin University.

Kingston, J.H. 1990.
"Algorithms and Data Structures: Design, Correctness, Analysis"
Addison-Wesley, Sydney.

Knuth, D.E., 1973.
"The Art of Computer Programming, Vol. 3: Sorting & Searching"
Addison-Wesley, Reading, Mass.

Mehlhorn, K., 1984.
"Data Structures & Algorithms, Vol. 1: Sorting & Searching"
Springer-Verlag, Berlin Heidelberg 1984

Moffat, A., Petersson, O., 1992.
"An Overview of Adaptive Sorting",
The Australian Computer Journal, Vol. 24, No.2, May 1992. pp. 70-77.

Sedgewick, R., 1980.
"Quicksort."
Garland Publishing Inc., New York & London.

Williams, J.W.J., 1964.
"Algorithm 232: Heapsort."
Communications of the A.C.M. Vol. 7, p. 701.

Wirth, N., 1976.
"Algorithms + Data Structures = Programs'
Prentice-Hall, Inc. Englewood Cliffs, N.J.