Formulation and analysis of in-place MSD radix sort algorithms

Nasir Al-Darwish

ICS Department, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia

Received 15 December 2004
Revised 25 April 2005

Abstract.

We present a unified treatment of a number of related in-place MSD radix sort algorithms with varying radices, collectively referred to here as 'Matesort' algorithms. These algorithms use the idea of in-place partitioning which is a considerable improvement over the traditional linked list implementation of radix sort that uses O(n) space. The *binary Matesort* algorithm is a recast of the classical radix-exchange sort, emphasizing the role of in-place partitioning and efficient implementation of bit processing operations. This algorithm is O(k) space and has O(kn) worst-case order of running time, where k is the number of bits needed to encode an element value and n is the number of elements to be sorted. The binary Matesort algorithm is evolved into a number of other algorithms including 'continuous Matesort' for handling floating point numbers, and a number of 'general radix Matesort' algorithms. We present formulation and analysis for three different approaches (sequential, divide-and-conquer and permutation-loop) for partitioning by the general radix Matesort algorithm. The divide-and-conquer approach leads to an elegantly coded algorithm with better performance than the permutation-loop-based American Flag Sort algorithm.

Correspondence to: Nasir Al-Darwish, ICS Department, King Fahd University of Petroleum and minerals, Dhahran, Saudi Arabia. E-mail: darwish@kfupm.edu.sa.

Keywords: sorting algorithms; radix sort; MSD radix sort; Matesort; general radix Matesort; Quicksort

1. Introduction

Sorting is a fundamental problem in computer science, with wide applications [1]. The moment the algorithm run-time and space complexity were formalized, sorting algorithms were at the forefront for analysis and improvement. Early recognition was given to Quicksort [2] and Heapsort [3] algorithms, because they improve on the O(n^2) order of running time of other slow sorting algorithms, to O(n log n). Heapsort is O(n log n) in the worst case, while Quicksort is O(n log n) in the average case. In practice, Quicksort runs faster than Heapsort most of the time, although it exhibits O(n^2) worst-case order of running time, which happens, for example, when the input data is nearly sorted. In general, sorting algorithms can be divided into two categories: 'comparison-based' and 'distribution-based'. A comparison-based algorithm, like Heapsort or Quicksort, sorts by comparing two elements at a time. On the other hand, a distribution-based algorithm, like radix sort [4–7], works by distributing the elements into different piles based on their values.

Radix sort algorithms fall into two classes: MSD (most significant digit) and LSD (least significant digit). Radix sort algorithms process the elements in stages, one digit at a time. A digit is a group of consecutive bits with the digit size (number of bits) set at the beginning of the algorithm. MSD radix sort starts with the most significant (leftmost) digit and moves toward the least significant digit. LSD radix sort does it the other way. LSD distributes the elements into different groups – commonly known as 'buckets' and treated as queues (first-in-first-out data structure) – according to the value of the least significant (rightmost) digit. Then the elements are re-collected from the buckets and the process continues with the next digit. On the other hand, MSD radix sort first distributes the elements according to their leftmost digit and then calls the algorithm recursively on each group. MSD needs only to scan distinguishing prefixes, while all digits are scanned in LSD. For example, for radix-2 MSD (i.e. digit size = 1 bit), two buckets are used and the elements are distributed into either bucket depending on the value of the most significant bit. Then the process continues with the next bit, considering only groups that have more than one element, until all the bits have been scanned. Clearly, such an algorithm uses O(n) space for the combined two buckets and is able to sort n non-negative integers in the range [0,m] in O(n log m) order of running time.

1.1. Previous related work

Historically, MSD and LSD radix sort algorithms have been implemented using O(n) space, either using a working array of size n or linked lists. A notable exception is 'radix exchange' sort [4] and further generalization by McIlroy et al. [7]. Radix exchange sort was first suggested for binary-alphabet but can be used with strings provided that bit-extraction and testing are done as low-level machine operations. The basic idea of radix exchange sort is to split *in-place* the data into two groups based on the most significant bit. This is done using two oppositely moving pointers; the left (right) pointer skips elements having 0-bit (1-bit); otherwise, it *exchanges* the elements pointed to by left and right pointers. Then the process is applied recursively to each group considering the next bit. Radix exchange sort is best thought of as a 'mating' of Radix sort and Quicksort, since in-place partitioning is a characteristic of Quicksort. Therefore, the author suggests that it be called 'Matesort'. Section 2 presents a formal treatment of radix exchange sort (i.e. binary Matesort) summarizing key theoretical and experimental results. Radix exchange sort has been unfairly upstaged by Quicksort despite the fact that it is simple to implement, runs fast, and has a worst-case order for running time lower than that of Quicksort. Worse still, many textbooks on algorithms fail to present it or even reference it. In Section 5, it is shown that binary Matesort algorithm works well for strings too.

To in-place implement a general radix (i.e. digit size larger than 1 bit) MSD radix sort, McIlroy et al. [7] proposed the 'American Flag' in-place partitioning method – named as such because the problem of in-place k-way partitioning is a generalization of the Dutch Flag (three-way in-place partitioning) problem [8, 9] proposed by Dijkstra. This uses the concept of a 'permutation loop' – it uses a preprocessing step to determine the count of elements (based on the current digit) that should belong to a particular digit value and this information in turn is used to determine where the element should be placed in the 'sorted on the current digit' array. In [7] it was concluded that the American Flag sort is the fastest for sorting strings – a conclusion confirmed by others [10]. In Section 4.3, we present analysis and optimized implementation of the permutation loop concept. However, this algorithm is found to be inferior to our proposed divide-and-conquer partitioning algorithm.

Apparently the work of McIlroy et al. [7] on in-place radix sort has received little attention or was found incomprehensible by some. The work reported here, which can be viewed as an improvement on, and extension to, their work, counters prevailing beliefs that MSD is complex to implement and would require lots of space. To quote: 'Most significant digit (MSD) Radix sort takes a lot more bookkeeping – the list must repeatedly be split into sublists for each value of the last digit processed' [11]; and 'The bookkeeping would quickly get out of hand; pointers indicating where the various buckets begin and information needed to recombine the elements into one list would have to be stacked and unstacked often' [12, p. 202].

More recent work on the subject of in-place radix sorting is reported in [13] and [14]. The proposed algorithms appear to use the permutation-loop idea; however, the presentations of these algorithms are sketchy at best and the authors fail to recognize and relate to earlier work by McIlroy et al. In [13], it is suggested that the digit size be 'adaptive' and varied during the execution of the algorithm. However, we have found no significant differences in the running times as the digit size is varied.

This paper presents a comprehensive, unified and readable treatment of in-place MSD radix sort. The rest of the paper is organized as follows. Section 2 presents the binary Matesort algorithm and compares its performance to other popular sorting algorithms. Section 3 presents continuous Matesort for sorting real numbers. In Section 4 we present several general radix Matesort algorithms including the new GenMatesort_DC (using divide-and-conquer partitioning) and establish a number of performance-related lemmas. The analysis confirms that for random data, general radix Matesort is no better than binary Matesort. To our knowledge, such an analysis has not been published before. Section 5 presents comparison results of five Matesort algorithms used for sorting English text. In this paper, we give complete program code listings in C# (also, equivalent to Java). This should enable easy verification of the claimed results and avoid any implementation ambiguity – as McIlroy et al. [7] suggested, 'The troubles with radix sort are in implementation, not in conception'.

2. Binary Matesort algorithm

The binary Matesort algorithm has some striking resemblance to Quicksort – the difference is in the extra 'bitloc' (bit location) input parameter and the partition method used. The idea for Matesort comes from 'algorithm design by induction' [15] using the following induction step.

Induction step: suppose any of the elements in the array $A[1 \ldots n]$ is encoded using k bits ($b_{k-1}\ b_{k-2} \ldots b_0$) and that $A[1 \ldots n]$ is partitioned based on the most significant bit (b_{k-1}) such that all elements with $b_{k-1} = 0$ appear before elements with $b_{k-1} = 1$. Then what remains to be done is to sort each group.

Theorem 1: the binary Matesort algorithm has $O(kn)$ worst-case order of running time, where n is the number of elements to be sorted and k is the number of bits needed to encode an element value. The algorithm is $O(k)$ space.

Proof: suppose that any of the elements is encoded using k bits ($b_{k-1}\ b_{k-2} \ldots b_0$). The running time for the BitPartition algorithm for n elements is easily shown to be $O(n)$ since it performs $O(1)$ (constant time) per element. The calls to Matesort can be depicted as a binary tree whose root is the initial call to Matesort. At any tree level, the array elements (n elements in total) are split disjointly among various calls to Matesort. The calls at level r (root at level 0 corresponds to $bitloc = k - 1$) are associated with calls to BitPartition with $bitloc = (k - r - 1)$. Thus the processing associated with any tree level is $O(n)$. The lowest tree level is for $bitloc = 0$ and thus the tree has a maximum of k levels. Thus the worst-case order of running time is $O(kn)$. The space used is dominated by the stack space associated with recursive calls (i.e. return addresses, call parameters and local variables). Each such call uses $O(1)$ space – note that BitPartition is $O(1)$ space. In the worst case, there may be k calls pending. Thus the algorithm is $O(k)$ space.

```
void Matesort(int[] A, int lo, int hi, int bitloc)
{ // initial call: bitloc = highest bit position (starting from 0)
    if ((lo < hi ) && (bitloc >=0))
    {   int k = BitPartition(A,lo,hi,bitloc);
        Matesort(A,lo,k,bitloc-1);
        Matesort(A,k+1,hi,bitloc-1);
    }
}

void Quicksort(int[] A, int lo, int hi)
{   if  (lo < hi )
    {   int k = Partition(A,lo,hi);
        Quicksort(A,lo,k-1);
        Quicksort(A,k+1,hi);
    }
}

int BitPartition(int[] A, int lo, int hi, int bitloc)
{   int pivotloc = lo-1; int t;
    int Mask = 1<< bitloc;
    for(int i= lo; i<=hi ; i++)
    // if ( ((A[i]>> bitloc) & 0x1) ==0)
    if  (    (A[i] & Mask) <= 0)
    { // swap with element at pivotloc+1 and update pivotloc
        pivotloc++;
        t = A[i]; A[i] = A[pivotloc]; A[pivotloc] = t;
    }
    return pivotloc;
}

int Partition(int[] A, int lo, int hi)
{   int t; int   pivot = A[lo]; int pivotloc =lo;
    for(int i= lo+1; i<=hi ; i++)
        if  (A[i] <= pivot)
        { // swap with element at pivotloc+1 and update pivotloc
        pivotloc++;
        t = A[pivotloc]; A[pivotloc] = A[i]; A[i] = t;
        }
    // move pivot to its proper location
    A[lo] = A[pivotloc]; A[pivotloc] = pivot;
    return pivotloc;
}
```

Listing 1. Binary Matesort algorithm in comparison with Quicksort algorithm.

2.1. Partitioning

A key operation for Quicksort and Matesort algorithms is the rearrangement of the array elements using certain criteria such as around a pre-selected value. We limit our attention to in-place partitioning (i.e. without using any additional working arrays). The $O(n)$ partitioning algorithms presented here are not new – they are included for the presentation to be self-contained. For binary Matesort, the following problem needs to be solved.

BitPartition problem: given an integer array $A[1 \ldots n]$ and a bit position i, return k such that bit b_i in any of ($A[1], A[2], \ldots, A[k]$) $\leq 0 <$ bit b_i in any of ($A[k + 1], A[k + 2], \ldots, A[n]$).

On the other hand, Quicksort requires a solution to the following problem.

Partition problem: given an integer array $A[1 \,..\, n]$ and a pivot value X. Return k such that any of $(A[1], A[2], \ldots, A[k]) \le X <$ any of $(A[k + 1], A[k + 2], \ldots, A[n])$.

For either problem, we refer to k as the pivot location. Induction can be easily employed to come up with good solutions to the above problems.

Partitioning_Method 1
Induction step: assume the array $A[1 \,..\, n - 1]$ is partitioned such that any of $(A[1], A[2], \ldots, A[k]) \le X <$ any of $(A[k + 1], A[2], \ldots, A[n - 1])$ – k is the pivot location.

Now consider $A[n]$. If $A[n] > X$ then done (pivot location is unchanged); otherwise, since $A[n] \le X \Rightarrow A[n] < A[k + 1]$, we can swap $A[n]$ with $A[k + 1]$ and advance the pivot location to $k + 1$.

The solution to the BitPartition problem is similar to the above except that the criterion used to update the pivot location ($A[n] \le X$) is replaced by (b_i of $A[n] \le 0$).

The implementation of this algorithm is shown in Listing 1. Note that for BitPartition, we utilize a bit manipulation operation. To isolate the bit at location *bitloc*, one can use the expression ($A[i]$ >> bitloc) & 0x1. It does *shift right* the value $A[i]$ *bitloc* bits and then AND with the hexadecimal value '00 . . . 01'. However, a more efficient method is to use the expression ($A[i]$ & Mask), where Mask = 2^{bitloc} computed as a loop invariant. For the partition method used by Quicksort, the first element is chosen as the pivot and then, at the end, the pivot is swapped with $A[k]$ to put the pivot into its proper sorting position. Such a step is not warranted for BitPartition. Note that the initial call to Matesort is passed the location of the highest bit. This can be determined by a cumulative OR operation over all input elements and then determining the position of the leftmost bit value of 1.

Another solution to the Partition problem is based on the following induction step.

Partitioning_Method 2
Induction step: compare $A[1]$ with $A[n]$. There are four cases to consider:

$A[1] \le X < A[n] \Rightarrow k$ = Partition of $A[2 \,..\, n - 1]$.
$A[1] \le X \ge A[n] \Rightarrow k$ = Partition of $A[2 \,..\, n]$.
$A[1] > X < A[n] \Rightarrow k$ = Partition of $A[1 \,..\, n - 1]$.
$A[1] > X \ge A[n] \Rightarrow$ swap $A[1]$ with $A[n]$, k = Partition of $A[2 \,..\, n - 1]$.

The iterative implementation of this method – see Partition_Cont2 in Listing 4 – utilizes two pointers (*left* and *right*) that are set to first and last array locations, respectively. The left pointer moves toward the right skipping elements $\le X$, whereas the right pointer moves toward the left skipping elements $> X$.

The two partitioning methods were found to have comparable running times for random data.

Although Method 2 does $n/4$ expected number of swaps (vs. $n/2$ for Method 1), it makes *two* comparisons three-quarters of the time in order to eliminate one array element (vs. one comparison per element for Method 1).

2.2. Binary Matesort compared to Quicksort and Heapsort

First, we should note that all the reported experiments were run on a Pentium IV 2.6 MHz 512 MB RAM running Windows XP 2002. The programs were written, compiled and run (Release Build) using Microsoft's C# VS.Net 2003. For the generation of random integer data for this experiment, we have used the *fillarray* function given in Listing 2. This function fills the array with integers in range [0, *max*] using uniform random distribution, where $max = n/x$. This scheme is simply referred to as (U/x) and x can be thought of as a repetition factor. Table 1 summarizes the results, where a given timing entry is an average of five runs. Note that $(5FFFFFF)_{hex}$ = 100,6632,295 (i.e. about 100 million elements). The results show that Quicksort and Matesort run neck-and-neck and clearly outperform other algorithms including the Microsoft .Net built-in Array Sort.

2.3. Binary Matesort compared to Quicksort

Quicksort is known to suffer from two problems that cause performance degradation – see Table 2 – whilst Matesort seems to be free of these problems. The first problem is that the algorithm (i.e. repeated partitioning) is slow – in comparison with other sorting algorithms – when applied to small size data, especially for data that is nearly sorted or containing identical values. Table 2 clearly shows this behavior as the data

```
void fillarray(int[] A, int n)
{   int repfactor=4; //Vary Repetition factor as necessary
    Random r = new Random();
    int max = n/repfactor;
    for(int i= 1; i<=n ; i++)
        A[i]= r.Next(max);
}
```

Listing 2. Generation of integer test data.

Table 1
Matesort compared to other sorting algorithms for random uniformly distributed integer data.
Quicksort_Mod and Matesort_Mod are for modified versions that use Insertion sort for input size < 20

Size (n), in hex	Distribution	Execution time (ms)		
		1FFFFFF	3FFFFFF	5FFFFFF
.Net Sort	$U/1$	14750	30234	46375
Heapsort		57921	133178	218843
Quicksort		8039	16812	26343
Quicksort_Mod		7734	15906	24328
Matesort		7843	16203	25359
Matesort_Mod		6718	14125	22234
.Net Sort	$U/2$	14296	29937	46031
Heapsort		57593	134078	218765
Quicksort		7968	16632	25351
Quicksort_Mod		6671	13984	21515
Matesort		7328	15296	24062
Matesort_Mod		6890	14468	22765
.Net Sort	$U/4$	14203	29509	44828
Heapsort		57718	134140	220062
Quicksort		8062	16632	25742
Quicksort_Mod		6625	13843	21171
Matesort		6937	14531	22859
Matesort_Mod		7046	14765	23203

repetition factor is increased. Altering the pivot selection strategy does not help. Also, we have observed that the strategy of coupling Quicksort with Insertion Sort (when input falls to about 20 to 30 elements) was of little value in these situations (i.e. ~0.001 redundancy).

The second problem is also apparent from Table 2, which shows that recursion depth and execution time increase as the data repetition factor is increased. In the worst case, the recursion depth (and associated stack space) can grow as bad as O(n). The Quicksort algorithm can be modified easily using 'tail-recursion elimination' to limit the recursion depth [2]. Tail-recursion elimination means replacing a recursive call located at the end of a procedure body block by a loop. The trick to limit recursion depth to O(log n) is to eliminate the recursive call that has more than half of the elements.

```
void Quicksort_TRO(int[] A, int lo, int hi)
{   while (lo < hi)
    {   int k = Partition(A,lo,hi);
        if ((k-lo) > (hi-k))
            {   Quicksort_TRO(A,k+1,hi);
                hi = k-1;     }
        else
            {   Quicksort_TRO(A,lo,k-1);
                lo = k+1;     }
    }
}
```

Listing 3. Tail-Recursion Optimized Quicksort.

3. Using Matesort algorithm for continuous domain

A notable attempt to use the element value to determine the 'sorting' address is exemplified by 'proximity map' sort. It uses a hashing function to determine the vicinity where the element will be placed in the final sorting order. However, such algorithms are generally complex, and very often only suitable for certain kinds of data. Dobosiewicz [16] presented a sorting algorithm based on distributive partitioning. The algorithm sorts n real numbers by distributing them into n intervals of equal width. His algorithm runs in O(n) expected time and O(n log n) worst-case time. However, his algorithm does not avoid pair-wise element comparisons

Table 2
Recursion depth and execution time for binary Matesort (MS), Quicksort-Tail_Recursion_Optimized (QSTRO) and three versions of Quicksort. Input: $A[i]$ = random in $[0, n/x]$, for n = 1FFFFF (hex). The Quicksort versions used for Partition: Method 1, Method 2, Method 2 with Pivot = median of A[lo], A[hi] and A[(lo+hi)/2]. This latter Partition method is used for QSTRO.

x: (U/x) Distribution	Recursion depth					Execution time (ms)				
	MS	Quicksort			QSTRO	MS	Quicksort			QSTRO
1	25	64	72	57	17	7843	8156	7953	8078	7890
125	25	204	247	222	13	6859	15437	9531	9421	9953
250	25	351	426	399	13	6812	24093	11984	11484	12812
500	25	619	811	730	12	6781	38656	16921	15750	18750
1000	25	1166	1567	1372	12	6734	69890	27406	24796	30968

altogether, since it requires the median of the elements to be computed. The algorithm we present next is much simpler and runs fast all the time since it does reduction at the bit representation level.

Upon careful examination of the binary Matesort algorithm, it can be observed that the first call to Bit-Partition, which considers the most significant bit (kth bit), is equivalent to partitioning around a pivot value of 2^k. Then, the call to BitPartition on the values $\leq 2^k$, uses a pivot value of 2^{k-1} and the call to BitPartition on the values $>2^k$, uses a pivot value of $2^k + 2^{k-1}$. This suggests using a version of Partition that is passed the pivot value as an input parameter and then recasting Matesort into Matesort_Cont – see Listing 4 – with two extra input parameters: a lower limit (initially set to 0 for non-negative data) and a pivot (initially set to half the maximum value among all elements). When the pivot value reaches 1 the data is sorted based on the integer parts of the input numbers. If we continue to halve the pivot further, we will be able to sort numbers that fall within 0.5 of each other, and so on. The process should continue until some minimum pivot value (fraction) is reached that is smaller than the precision of the input data.

It may be considered annoying to set a minimum pivot value before calling continuous Matesort. Alternatively, the process of halving the pivot can continue until an identical elements condition is reached. Matesort_ContV2 implements this idea. The test for identical elements is incorporated within the Partition method – see commented lines in Partition-Cont2 in Listing 4 – by having the method return –1 if all elements are equal. As shown in Table 3, this version of continuous Matesort runs slightly faster than the other version. The table shows the results of some tests (data generated using the *fillarray* method given in Listing 4 where an element has a fractional part with 0.001 accuracy) carried out to compare continuous Matesort, Quicksort and Microsoft .Net built-in Array Sort. The test data varied in size between 10 million

Table 3
Execution times for .Net built-in Sort, Quicksort and Continuous Matesort.
Quicksort uses for Partition: Method 2 with Pivot = median of A[lo], A[hi] and A[(lo+hi)/2].
Entries in parentheses are for versions of these algorithms coupled with Insertion sort for input size < 20.

Array size (n) $\times 10^6$	Execution time (ms)			
	.Net Sort	Quicksort	Matesort_Cont	Matesort_ContV2
10	7362	3343 (2875)	4281 (2812)	4015 (2765)
20	14843	6984 (6046)	8781 (5843)	8296 (5796)
40	31109	14500 (12953)	17890 (12187)	17203 (12125)
50	39234	18343 (16355)	22609 (15593)	22362 (15437)

```
void Matesort_Cont(double[] A,int lo,int hi,double fromval,double inc)
{   if ((lo < hi ) && (inc >=minpivot))
    {   int k= Partition_Cont2(A,lo,hi,fromval+inc);
        Matesort_Cont(A,lo,k, fromval, inc/2.0);
        Matesort_Cont(A,k+1,hi, fromval+inc, inc/2.0);
    }
}

void Matesort_ContV2(double[] A,int lo,int hi,double fromval,double inc)
{   if (lo < hi)
    {   int k= Partition_Cont2(A,lo,hi,fromval+inc);
        if (k==-1) return;
        Matesort_ContV2(A,lo,k, fromval, inc/2.0);
        Matesort_ContV2(A,k+1,hi, fromval+inc, inc/2.0);
    }
}

int Partition_Cont2(double[] A, int lo, int hi, double pivot)
{   double t; int i,j;
    //  return –1 if all elements are equal; enable for Matesort_ContV2
    //  for(i=lo; i< hi; i++)
    //      if (A[i] != A[hi]) goto cont;
    //  return -1;
    //  cont:
            i =lo; j=hi;
            while (i < j)
            {   while ( (i <=hi) && ( A[i] <= pivot )) i ++;
                while ( (j >= lo) && ( A[j] > pivot )) j --;
                if (i < j)
                    {   t = A[i]; A[i] = A[j]; A[j] = t;   }
            }
        return j;
}

void fillarray(double[] A, int n)
{   Random r = new Random();
    for(int i= 1; i<=n ; i++)
        A[i]= r.Next(n) + r.Next(1000)/1000.0;
}
//-- main
    int n= 40000000; double[] D = new double[n+1];
    fillarray(D,n);
    minpivot=.0005; //Set Minimum Pivot before calling Matesort_Cont.
    Matesort_Cont(D,1,n,0,n/2);
```

Listing 4. Matesort algorithms for real numbers.

and 50 million elements. A given timing entry is an average of five runs. Although the results show that Quicksort and Matesort have comparable speed, one has to remember that the given Matesort algorithms when run for *n* numbers with a magnitude precision of *k* bits, have worst-case running time of $O(kn)$ vs $O(n^2)$ for Quicksort. The results show that Microsoft .Net built-in Array is much slower (50% slower) than either Matesort or Quicksort.

4. General radix Matesort algorithms

Whereas the basic Matesort algorithm processes the data one bit at a time, the general radix Matesort – see Listing 5 – processes the data one digit (a group of bits) at a time. A radix value of *r* implies that the digit size = log *r*, e.g. radix 16 uses a digit size of 4 bits. Thus, compared to binary Matesort, we observe the following. First, the BitPartition method is replaced by the

DigitPartition method that is passed *digitloc* (digit location) and *digitval* (digit value) parameters. Second, for radix *r* and a given *digitloc*, the data must be split into *r* groups, one for each radix value. This process can be done sequentially, one digit value at a time (sequential partitioning), or through divide-and-conquer. For sequential partitioning (see GenMatesort_Seq in Listing 5), we need to issue exactly $(r - 1)$ calls to DigitPartition to get the data split into *r* parts and then issue *r* recursive calls to GenMatesort_Seq, one for each part, to process the data for the next digit location (*digitloc* – 1).

4.1. An elegant GenMatesort_DC

There is a potential for inefficiency in sequential partitioning. Namely, it is possible that after scanning all elements for the purpose of partitioning around the first digit value, we find no element ≤ *pivot*, and the process repeats on the whole set of input elements (*n*) for subsequent digit values. Thus it is possible that sequential partitioning may degenerate into O(*rn*) running time per digit location. A better partitioning strategy is to partition around the middle digit value (*mid*). This way the elements that fall below (above) *mid* are out of consideration when DigitPartition is called for digit values > *mid* (digit values ≤ *mid*). If the data is random and uniformly distributed then this

```
int DigitPartition(int[] A,int lo,int hi,int digitloc,int pivot)
{   int t, int bitloc = 4*digitloc; // Multiplier is Radix dependent
    int pivotloc=lo-1;
    for(int i= lo; i<=hi ; i++)
        if ( ((A[i]>> bitloc) & 0xF) <=pivot ) // Mask is Radix dep.
            { pivotloc++; t = A[pivotloc]; A[pivotloc] = A[i];   A[i] = t; }
    return pivotloc;
}

// Note: Initial call is GenMatesort_Seq(A,1,n,digitcount-1)
void GenMatesort_Seq(int[] A,int lo,int hi,int digitloc)
{   int digitval, k;
    if ((lo < hi ) && (digitloc >=0))
        {   for(digitval=0; digitval < 15; digitval++) // Range is Radix dep.
            {   if (lo>=hi) break; // optionally added for optimization
                k = DigitPartition(A,lo,hi,digitloc,digitval);
                GenMatesort_Seq(A,lo,k, digitloc-1);
                lo = k+1;
            }
            GenMatesort_Seq(A,lo,hi, digitloc-1);
        }
}

// Note: Initial call is GenMatesort_DC(A,1,n,digitcount-1,0,15)
void GenMatesort_DC(int[] A,int lo,int hi,int digitloc,int fromdigitval, int todigitval)
{   int mid, k;
    if ( (lo < hi ) && (digitloc >=0) )
        if (fromdigitval < todigitval)
        {   mid = (fromdigitval +todigitval)/2;
            k= DigitPartition(A,lo,hi,digitloc,mid);
            GenMatesort_DC(A,lo,k, digitloc, fromdigitval, mid);
            GenMatesort_DC(A,k+1,hi, digitloc, mid+1, todigitval);
        }
        else GenMatesort_DC(A,lo,hi, digitloc-1,0, 15); // Range is Radix dep.
}
```

Listing 5. General radix Matesort using sequential partitioning (GenMatesort_Seq) and general radix Matesort using divide-and-conquer Partitioning (GenMatesort_DC) – versions given are for Radix = 16.

results in $O(n \log r)$ running time for the first digit – see Lemma 2 below.

Our original attempt to code the divide-and-conquer idea was to use a method that returns an array of pivot locations for pivot values ranging over all possible digit values. Upon reflection, we realized that such a method serves no purpose other than organizing and sequencing the calls to DigitPartition. Can we not simply use GenMatesort_DC itself for that? The answer is, of course, 'Yes', if we extend the parameters of GenMatesort_DC to include *fromdigitval* and *todigitval* parameters. Thus the six line (literally a single *if* statement) algorithm given in Listing 5 was born. Note that for a given *digitloc* GenMatesort_DC is repeatedly called (first two recursive calls) until we reach a single digit value (*fromdigitval* < *todigitval* is false). At that point, we issue a call GenMatesort_DC (resetting the *fromdigitval*, *todigitval* range) to continue with the next digit location (*digitloc* – 1).

4.2. Lemmas and performance results for general radix Matesort

For n elements data and radix r, assuming the data is random and uniformly distributed, the following lemmas hold, assuming partitioning is performed using Partitioning_Method1 given in Section 2. Also, note that element access count does not include element access during a swap operation.

Lemma 1: for random uniformly distributed data, the expected number of element accesses [swaps] performed by the general radix Matesort using sequential partitioning for the first digit location is $(r-1)\, n - (n/r)(r-2)(r-1)/2$

[recurrence equations: $C(n, 1) = 0$; $C(n,r) = (r-1)(n/r)/r + C(n - n/r, r-1)$ for $r > 1$].

Proof: the first call to DigitPartition examines n elements (using the first radix value as pivot) and the data is split into two parts of length n/r and $(n - n/r)$. DigitPartition is then called on the second part (using the second radix value as pivot), etc. Thus,

# of elements accessed = $n + (n - n/r) + (n - 2n/r) + \ldots$, for a total $r-1$ terms.
$= (r-1)n - n/r\,(1 + 2 + \ldots + r - 2) = (r-1)n - (n/r)\,(r-2)(r-1)/2$.

For $r = 256$, # of elements accessed = $255\, n - (n/256)(127{*}255) \approx 128\, n$.

Now consider swaps. For the first call to DigitPartition, the following takes place. Since n/r elements are expected to fall within the first digit value and each of the r parts contributes equally [i.e. each part contributes $(n/r)/r$ elements]. Thus $r-1$ of the parts will contribute a total of $(r-1){*}\,(n/r^2)$ elements via swaps. Thus,

# of swaps = $(r-1){*}\,(n/r)/r + (r-2) {*} [(n - n/r)/(r-1)] + \ldots$, for a total of r-1 terms.

This is equivalent to the recurrence: $C(n,1) = 0$; $C(n,r) = (r-1)(n/r)/r + C(n - n/r, r-1)$ for $r > 1$.

Lemma 2: for random uniformly distributed data, the expected number of element accesses [swaps] performed by the general radix Matesort using divide-and-conquer partitioning for the first digit location is $(r-1)n\ [n/2 \log r]$.

Proof: the first call to DigitPartition examines n elements and the data is split into two equal parts for two subsequent calls. Then these two parts are examined and each part generates two subsequent calls, etc. Thus,

# of elements accessed = $n + 2 {*} (n/2) + 4 {*} (n/4) + \ldots$, for a total of $\log r$ terms.
$= n \log r$.

For $r = 256$, # of elements accessed = $8n$.

For swaps, it is expected that one half of the elements for a particular part comes from the other part via swaps. Thus during the first call to DigitPartition, $(n/4 + n/4)$ swaps are made. Then each of these two parts is associated with two subsequent calls to DigitPartition, etc. Thus,

# of swaps = $2{*}\, n/4 + 4 {*} (n/8) + 8 {*} (n/16) + \ldots$, for a total of $\log r$ terms.
$= (n/2) \log r$.

For $r = 256$, # of element swaps = $4n$.

How about formulas for Binary Matesort?

Binary Matesort can be considered as sequential and divide-and-conquer at the same time, with a digit size of 1 bit ($r = 2$). Thus, we can substitute $r = 2$ in the formulas given in Lemma 1 or Lemma 2.

Lemma 3: for random uniformly distributed data, the expected number of element accesses [swaps] performed by binary Matesort for the first bit location is $n[n/2]$.

Now, for random uniformly distributed data, following the partitioning on the first digit location, we end up with r sets, each of size n/r, that are to be partitioned on the second digit location. If we assume that the elements in each set are still random and uniformly distributed with respect to radix values, then Lemma 4 is justified.

Lemma 4: for random uniformly distributed data, let $C(n,r)$ denote the count given by any of the above lemmas, then the total count over all digit locations, $TC(n,r,k)$, assuming each element is encoded using k digits, is $C(n,r) + rC(n/r,r) + r^2C(n/r^2,r) + \dots + r^{m-1} C(n/r^{m-1},r)$, where m is minimum of $\{k-1$, maximum j such that $r^j \leq n\}$. The per digit count is $TC(n,r,k)/k$.

Using Lemmas 3 and 4 to count the swaps made by binary Matesort for the leftmost 8 bits, we get $n/2 + 2 * (n/4) + \dots$ (8 terms) $= 4n$. This is consistent with the numbers obtained for binary Matesort given in Table 4. Likewise, the number of element accesses made by binary Matesort for the leftmost 8 bits is $8n$. Comparing these numbers to those of divide-and-conquer partitioning confirms that these algorithms have comparable performance.

Table 4 shows a summary of results for an experiment that was run to gauge how the above formulas correlate with the computed (via impeded counting statements) counts. The numbers in parentheses are for (*computed*, *formula*), in that order, for the first digit location (i.e. 8-bit character). The results are consistent with the above lemmas. Table 4 shows the element access count per character location (computed total divided by character count) and element swap count per character location. Also, note that – consistent with Lemma 4 – the counts get smaller and smaller for subsequent character locations and lead to a per character average that is *smaller* than that computed for the first character location. For presentation clarity, we have eliminated the per character location count as given by the formula in Lemma 4. It can be easily verified that the formula in Lemma 4 gives a good approximation. For example, evaluating $TC(n,r,k)/k$ for $n = 1000000$, $r = 256$, $k = 20$ and $C(n,r)$ given by Lemma 1, gives 19,274,414 which is a close approximation to the computed 19,656,858 (4th entry in GR MS_Seq row).

4.3. *General radix Matesort using multi-digit partitioning*

In general, partitioning a data array on a range of digit values can be done in one of two ways. The first way is to organize a sequence of calls against a 'single value' digit partition algorithm. This is exactly what GenMateSort_Seq and GenMateSort_DC did. The second way is to generalize the digit partitioning algorithm itself to handle a set of multiple pivots and return an array of pivot locations. We refer to any such partitioning algorithm as *multi-digit partitioning*. We have found that generalization of Partitioning_Method1 to handle multiple pivots produces element access and swap counts comparable to sequential partitioning but

Table 4
Count of element accesses and swaps (per character and for leftmost character) for various Matesort algorithms. Input: random (byte value [0,255]) string arrays, string length=20.

Array size $\times 10^3$	Execution time (ms)		Count of element accesses per char loc (compound, formula), for first char loc		Count of element swaps per char loc (computed, formula), for first char loc	
	100	1000	100	1000	100	1000
Binary MS	109	1865	89708, (800000, same)	1063240, (8000000, same)	44821, (399647, 400000)	531324, (4001032, 4000000)
GR MS_Seq	1031	14093	1788803, (12858966, 12851211)	19656858, (128561007, 128496093)	13891, (99571, 97607)	152302, (996010, 976076)
GR MS_DC	109	1875	89747, (800000, same)	1063129, (8000000, same)	44872, (399611, 400000)	531520, (3998714, 4000000)
GR MS_MDPLoop	234	2265	410143, (199884, 199609)	1501925, (1996395, 1996093)	11282 (99628, 99609)	139360, (996139, 996093)

with twice the running time – most probably due to access of arrays used to keep information about each digit. A more efficient algorithm is to use a divide-and-conquer approach. Following a similar analysis to that of Lemma 1 and Lemma 2 shows that, to partition n elements around r pivots, the count of element accesses for these two approaches is $nr/2$ and $n \log r$, respectively. Next, we consider the 'permutation-loop' algorithm introduced by McIlroy et al. [7] for the so-called American Flag sort algorithm.

The 'permutation loop' algorithm uses a preprocessing step to determine the count of elements (based on the current digit) that belong to a particular digit value and this information in turn is used to determine where the element should be placed in the 'sorted on the current digit' array. Then looping through all elements, for each element, x, let px = SortPosition of (x); swap(x,A[px]). Now x is in its proper position but A[px] may not be, so the process continues with A[px]. The implementation of this idea is given by the MultiDigitPartition_PLoop method shown in Listing 6.

It can easily be argued that MultiDigitPartition_PLoop on n elements performs at least $2n$ element accesses (n for the loop that does the counting, another n for the loop that puts the elements in their proper places) and at most $n - 1$ element swaps (because each swap puts at least one element in its place). The next lemma gives tighter limits for the first digit location.

Lemma 5: for random uniformly distributed data, the expected number of element accesses [swaps] performed by MultiDigitPartition_PLoop for the first digit location is $n + n(r - 1)/r$ [$n(r - 1)/r$].

Proof: consider element swaps first. A swap is needed for any element that is out of place. The probability of an element being out of place is $(r - 1)/r$. Thus # of swaps = $n(r - 1)/r$.

Next, consider element access. Note that there are n accesses for the loop that does the counting. Then for the loop that places the element, the ith position may be accessed as many times as a swap is needed. This leads to $n + n(r - 1)/r$ element accesses.

How good is the Permutation-Loop Partitioning method?

Consider the count of element accesses [swaps] for the multi-digit permutation-loop method for the first digit location. Using Lemma 5 – also consistent with experimental numbers reported in Table 4, – we get, for $r = 256$, # of element accesses [swaps] $\approx 2n$ [n]. These counts are much smaller than those – found to be $8n$ [$4n$] – of divide-and-conquer that uses Partitioning_Method1 of Section II. But these numbers do not tell the whole story – we did not count other operations involving the arrays that keep information about each digit value. Accordingly, the running time of GenMateSort_MDPLoop was found to be larger than that of GenMateSort_DC – see Tables 5 and 6. This point can also be seen from another angle. If the permutation-loop method is so good why not use it for 'single value' partitioning (e.g. use it for radix = 2)? However, the number of element accesses [swaps] for the 'single value' version of the permutation-loop method for random uniformly distributed data is $n + n/2$ [$n/2$] – obtained by either following through the algorithm logic or substituting $r = 2$ in Lemma 5. This is larger than those of Partitioning_Method1 (see Lemma 3), where the count of element accesses [swaps] is n [$n/2$].

5. Using Matesort algorithms for text data

The results of the previous section have established that, for random data, general radix Matesort is no better than binary Matesort. However, as demonstrated here, general radix Matesort is able to exploit data (and encoding) redundancy – something that binary Matesort is oblivious to. For example, if we know beforehand that the text is limited to English upper-case letters, then, rather than scanning the whole range of values (0–255) represented by a byte, the range should be limited to 65–90 (corresponding to the ASCII encoding of letters 'A'–'Z'). Here we consider several Matesort algorithms: binary Matesort, GenMatesort_Seq, GenMatesort_MDPLoop, GenMatesort_DC and GenMatesort_DC_Opt. GenMatesort_MDPLoop is the same as given in Listing 6. Listing 7 shows how BitPartition (used by *binary Matesort*) and DigitPartition are adopted for text data. For BitPartition, the highest bit location (charcount*8–1) corresponds to the leftmost bit of the first string character. Similarly, for DigitPartition, the highest character location (charcount–1) corresponds to the first string character. We have observed that representing the data using array of strings rather than array of characters is faster. This is probably because array element assignment in the former case involves pointer movement, whereas, in the latter case, the statement 'char[] t = A[pivotloc]' is apparently doing character-by-character copying.

5.1. Exploiting data redundancy

As shown in Listing 9, GenMatesort_DC can use a modified division rule for the computation of *mid*,

```
void GenMateSort_MDPLoop(string[] A,int lo,int hi,int digitloc)
{   int digitval, k;
    if ((lo < hi ) && (digitloc >=0))
    {   int[] splitloc = MultiDigitPartition_PLoop(A, lo,hi,digitloc);
        for(digitval=0; digitval< 255; digitval++) // range depends on radix
            {   k= splitloc[digitval];
                if (k==-1) continue;
                GenMateSort_MDPLoop(A,lo,k, digitloc-1);
                lo = k+1;
            }
        GenMateSort_MDPLoop(A,lo,hi, digitloc-1);
    }
}

int[] MultiDigitPartition_PLoop(string[] A,int lo,int hi,int charloc)
{   string t; int grp ,sumcnt,pi, lastgrp=0;
    int[] endloc= new int[256]; int[] startloc= new int[256];
    int[] count= new int[256]; //overloaded:also used for currenntloc in a group

    charloc=charcount-1-charloc; //added for order fixing for text data

    for(int i=0;i<=255;i++) { endloc[i]=-1;count[i]=0;  }
    //distribute elements based on value of current digit (char)
    for(int i= lo; i<=hi ; i++) count[A[i][charloc]]++;

    sumcnt = lo-1;
    for(int i=0;i<=255;i++)
        if (count[i]> 0)
        {   startloc[i] = sumcnt+1;
            sumcnt = sumcnt+count[i];
            endloc[i] = sumcnt;
            count[i] = startloc[i]; //reuse count for currentloc
            lastgrp=i;
        }

//optimize: hi is reduced by size of last group
hi = startloc[lastgrp]-1;

for(int i= lo; i<=hi ; i++)
    while (true)
    {       grp =A[i][charloc] ;
            if ((i >= startloc[grp]) && (i<=endloc[grp])) break;
        // get sort position for A[i]
        //  pi = count[grp]; count[grp]++; //optimize: replaced by next loop
            while(true)
            {   pi = count[grp]; count[grp]++;
                 if ( A[pi][charloc] !=grp ) break;
            }
            // swap A[i] with A[pi]
            t = A[pi]; A[pi]=A[i];    A[i] = t;
        }
    return endloc; //note:endloc of lastgroup not needed by caller
}
```

Listing 6. GenMateSort_MDPLoop, general radix Matesort using permutation loop partitioning – version given is for radix 256 used for text data.

```
int BitPartition(string[] A, int lo, int hi, int bitloc)
{   string t; //next 2 lines: map bitloc to charloc and bitloc within char
    bitloc=charcount*8<\#208>1-bitloc;
    int charloc=bitloc / 8;
    bitloc = 7 - (bitloc % 8);
    int Mask = 1<< bitloc ;
    int pivotloc =lo-1;
    for(int i= lo; i<=hi ; i++)
        // if ( ((A[i][charloc] >> bitloc) & 0x1) ==0 )
            if ( (A[i][charloc] & Mask) <= 0 )
            {   pivotloc++;
                t = A[i];   A[i] = A[pivotloc]; A[pivotloc] = t;
            }
    return pivotloc;
}

int DigitPartition(string[] A, int lo, int hi, int charloc, int pivot)
{   string t;
    charloc=charcount-1-charloc; //order fix: remap charloc
    int pivotloc=lo-1;
    for(int i= lo; i<=hi ; i++)
        if ( A[i][charloc] <= pivot )
        {   pivotloc++;
            t = A[pivotloc];    A[pivotloc] = A[i];            A[i] = t;
         }
    return pivotloc;
}

void GenMatesort_DC(string[] A,int lo,int hi,int digitloc,int fromdigitval, int todigitval)
{   int mid, k;
    if ((lo < hi ) && (digitloc >=0) )
        if ( fromdigitval < todigitval )
        { // mid = (fromdigitval +todigitval)/2;
            mid = (7*fromdigitval +3*todigitval)/10; // seven-three rule
            k = DigitPartition(A,lo,hi,digitloc,digitval);
            GenMatesort_DC(A,lo,k, digitloc, fromdigitval,mid);
            GenMatesort_DC(A,k+1,hi, digitloc,mid+1,todigitval);
        }
        else GenMatesort_DC(A,lo,hi, digitloc-1,64, 90);
    // else GenMatesort_DC(A,lo,hi, digitloc-1,0, 255);
}
```

Listing 7. BitPartition and DigitPartition adopted for text data (string array). Also shown, GenMatesort_DC using the seven–three rule and restricted range (64–90); commented lines are for unrestricted range.

such as ‘mid=(x*fromdigitval+(10–x)*todigitval)/10’ for some integer x between 1 and 9. In general, a good division rule is one that partitions the input data into two nearly equal-size halves. After some investigation, it was observed that the seven–three division rule produced timing results that were about 15% less than those produced by the normal (i.e. $x = 5$) division rule. However, we have omitted these figures because they remain larger (albeit slightly) than the figures obtained for GenMatesort_DC_Opt. There is a simple explanation why the seven–three rule works better than the normal rule. Since the letters at the lower end of the English alphabet occur more frequently than the letters at the higher end (e.g. the letters ‘A’–‘D’ are more frequent than the letters ‘W’–‘Z’), the division point should be moved closer toward the lower end in order to have an approximately 50–50 split of the data and the seven–three rule does exactly that. With the seven–three rule applied to the '@'–'Z' range, we get the two groups '@'–'H' and 'I'–'Z'. Clearly, considering letter frequency again, the 'I'–'Z' group is expected to be more evenly split with the seven–three rule than the normal rule.

Rather than using a fixed division rule, the mid digit may be determined based on actual character frequency but this was found to be slow. We have opted for an approximate solution. Listing 8 shows GenMatesort_DC_Opt. This is basically GenMatesort_DC modified to compute the *exact* endpoints of the *fromdigitval-todigitval* range by a process of scanning for minimum and maximum digit (character) values. This is only done when a particular digit location is visited for the first time for a given *lo-hi* group of elements. During such a scan, the digit values are summed and the average is used as the *mid* value at that time. As shown in Table 5, this produced the fastest timing results.

For the timing results shown in Table 5, the input data consisted of one million fixed length strings (with lengths fixed at 20, 30, 40, and 50 characters) extracted from English documents. A string is formed from the upper-case letters: 'A' to 'Z' plus '@' character – used as replacement for the blank character because its ASCII code (64) immediately precedes the ASCII code for the letter 'A'.

6. Concluding remarks

In this paper, we have presented and analyzed a number of in-place MSD radix sort algorithms, collectively referred to as Matesort algorithms. These algorithms are evolved from the classical radix exchange sort. Experiments have shown that binary Matesort is of comparable speed to Quicksort for random uniformly distributed integer data. Unlike Quicksort, which becomes slower as data redundancy increases

```
void GenMateSort_DC_Opt(string[] A,int lo,int hi,int digitloc,int fromdigitval,int todigitval)
{   int mid, k; int i,charloc; int c; int sum;
    if ((lo < hi) && (digitloc >=0) )
    {   sum =0;
        if (fromdigitval==0) // set fromdigitval,todigitval range
        {   charloc = charcount-1-digitloc; //added for order fixing for text
            fromdigitval = A[lo][charloc]; todigitval = fromdigitval;
            for (i=lo+1;i<=hi; i++)
                {   c = A[i][charloc];
                    sum = sum +c;
                    if (c < fromdigitval) fromdigitval = c;
                    else if (c > todigitval) todigitval = c;
                }
        }

    if ( fromdigitval < todigitval)
    {   if (sum > 0) // if range is set then set mid to average value
            mid= sum/(hi-lo);
        else mid = (fromdigitval +todigitval)/2;
    // else mid = (7*fromdigitval +3*todigitval)/10; // seven-three rule

            k= DigitPartition(A,lo,hi,digitloc,mid);
            GenMateSort_DC_Opt(A,lo,k, digitloc, fromdigitval,mid);
            GenMateSort_DC_Opt(A,k+1,hi, digitloc,mid+1, todigitval);
        }
    else GenMateSort_DC_Opt(A,lo,hi, digitloc-1,0, 255);
//In the statement above, fromdigitval parameter (0) is used as a flag to tell
//the start of examination of a digit location (digitloc) for this (lo,hi) group
    }
}
```

Listing 8. GenMateSort_DC_Opt: Optimized GenMateSort_DC to automatically do range restriction.

Table 5
Execution times for English text data. The times in parentheses are for restricted radix values to 27 characters (ASCII Codes 64–90). The second set of numbers in the last column is for the seven–three rule.

String array Size (n) = 10^6	Execution time (msec)						
String length	.Net Sort	Quick Sort	Binary Matesort	GR MS_Seq	GR MS_MDPLoop	GR MS_DC	GR MS_DC_OPT
20	6578	3593	6375	37859 (6656)	7390 (6343)	4921 (4828)	3468, 3281
30	6859	3718	9015	42859 (7640)	11546 (10140)	5890 (5781)	3640, 3421
40	7125	4109	10703	48973 (8765)	14890 (12406)	6953 (6890)	3828, 3625
50	7406	4531	12546	52593 (9703)	19593 (15968)	7890 (7718)	4046, 3828

and may degenerate into O(n^2) algorithm, binary Matesort algorithm is unaffected and remains bounded by O(kn), where k is the element size in bits.

We have discussed three partitioning methods for use by the general radix Matesort algorithm: sequential, divide-and-conquer and permutation-loop. For English text, experiments have shown that the general radix Matesort using divide-and-conquer partitioning is the fastest. Moreover, the divide-and-conquer method can be optimized further to exploit data redundancy. Finally, the experimental results reported here show that Matesort (and also Quicksort) are much faster (twice as fast in many instances) than the built-in Microsoft .Net Array Sort method, which suggests that Microsoft ought to examine their implementation.

References

[1] D.E. Knuth, *The Art of Computer Programming – Vol. 3 Sorting and Searching* (Addison-Wesley, Reading, MA, 1973).

[2] C.A.R. Hoare, Quicksort, *Computer Journal* 5(1) (1962) 10–16.

[3] J.W. Williams, Algorithm 232: Heapsort, *Communications of the ACM* 7(6) (1964) 347–8.

[4] P. Hildebrandt and H. Isbitz, Radix exchange – an internal sorting method for digital computers, *Journal of the ACM* 6(2) (1959) 156–63.

[5] I.J. Davis, A fast radix sort, *The Computer Journal* 35(6) (1991) 636–42.

[6] R. Sedgewick, *Algorithms* (Addison-Wesley, Reading, MA, 1988).

[7] P.M. McIlroy, K. Bostic and M.D. McIlroy, Engineering radix sort, *Computing Systems* 6(1) (1993) 5–27.

[8] E.W. Dijkstra, *A Discipline of Programming* (Prentice Hall, Upper Saddle River, 1997).

[9] C.L. McMaster, An analysis of algorithms for the Dutch National Flag Problem, *Communications of the ACM* 21(10) (1978) 842–6.

[10] A. Andersson and S. Nilsson, Implementing radixsort, *ACM Journal of Experimental Algorithmics* 3 (1998) article 7. Available at http://wotan.liu.edu/docis/dbl/acmexa (accessed 18 July 2005).

[11] D. Rig, *A Comparison of Sorting Algorithms* (2003). Available at www.devx.com/vb2themax/Article/19900 (accessed September 2004).

[12] S. Basse and A.V. Gelder. *Computer Algorithms* (Addison-Wesley, Boston, MA, 2000).

[13] A. Maus, ARL – a faster in-place, cache friendly sorting algorithm. In: N. Stol et al. (eds), *Norsk Informatik Koferranse NIK'2002, Kongsberg, 26 November 2002* (2002) 85–95. Available at www.nik.no/2002/ (accessed 18 July 2005).

[14] A. Al-Badaraneh and F. Al-Aker, Efficient in-place radix sorting, *Informatica* 15(3) (2004) 295–302.

[15] U. Manber, *Introduction to Algorithms: A Creative Approach* (Addison-Wesley, Reading, MA, 1989).

[16] W. Dobosiewicz, Sorting by distributive partition, *Information Processing Letters* 7(1) (1978) 1–6.