# The Practice of Using MSD Radix Sort Sorting Chinese Names

# and

# Survey of '*Formulation and analysis of in-place MSD radix sort algorithms*'

AN YU, College of Engineering, Northeastern University, USA
LAN GAO, College of Engineering, Northeastern University, USA

**Abstract.**
Radix sort algorithm, whose time complexity is $O(nlgn)$, is one of the most efficient sort methods, but it is fit for integer data with identical digits. To sort Chinese characters with the MSD Radix correctly and efficiently, we chose to convert the Chinese character array into a two-dimensional array before sorting and store the Chinese characters and the corresponding pinyin string. Experiments show that this conversion performed well in sorting Chinese names. This article also gives a brief introduction of '*Formulation and analysis of in-place MSD radix sort algorithms*' which is a published paper related to MSD radix sort.
Key words: Chinese string, Pinyin, MSD radix sort, ASCII

## 1 INTRODUCTION

### *1.1 What is MSD radix sort*

Most of the sort methods are usually used directly to sort integers, English words, and other kinds of type arrays with can be compare directly. So does MSD Radix sort. To implement a general-purpose string sort, where strings are not necessarily all the same length, we consider the characters in left-to-right order. We know that strings that start with a should appear before strings that start with b, and so forth. The natural way to implement this idea is a recursive method known as most-significant-digit-first (MSD) string sort [1]. And the cost of MSD radix sort is related to the number of possible characters in the alphabet.

### *1.2 How to use MSD radix sort to sort Chinese names*

we convert the Chinese character array into a two-dimensional array before sorting and store the Chinese characters and the corresponding pinyin string. Each sort directly operates on the pinyin string, so that the whole sort is basically converted to English character sort, radix can also use the 256 ascii code table. Time is reduced to linearithmic.

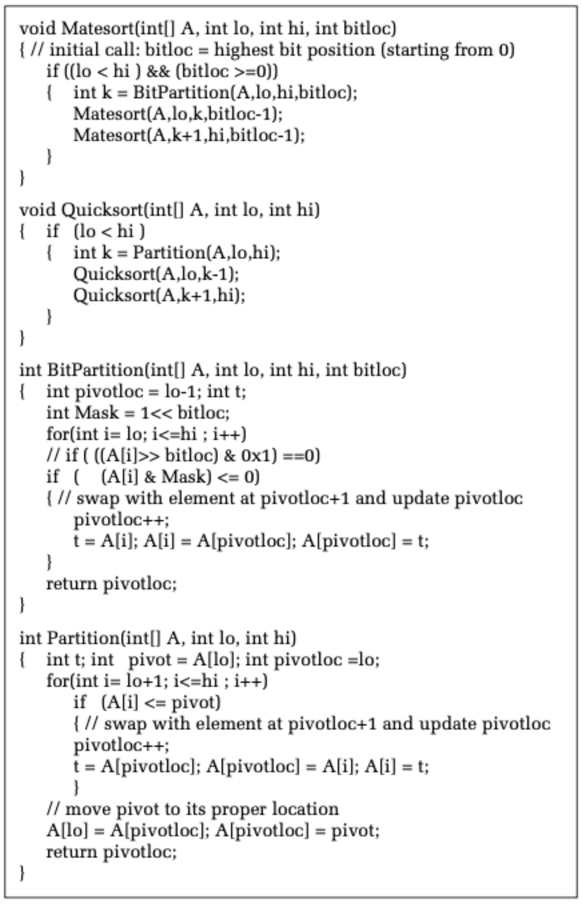
### *1.3 What is Matesort*

In '*Formulation and analysis of in-place MSD radix sort algorithms*', the author, Nasir Al-Darwish, came up with a unified treatment of a number of related in-place MSD radix sort with vary radices. 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'.

## 2 ALGORITHMS

### *2.1 Binary Matesort algorithm*

Binary Matesort has its own features in the extra bit location input parameter and partition method compared to Quicksort. For the worst-case order, running time and space used are $O\ (kn)$ and $O\ (k)$ respectively (any elements is encoded using k bits). The codes are showing in the Listing 1.



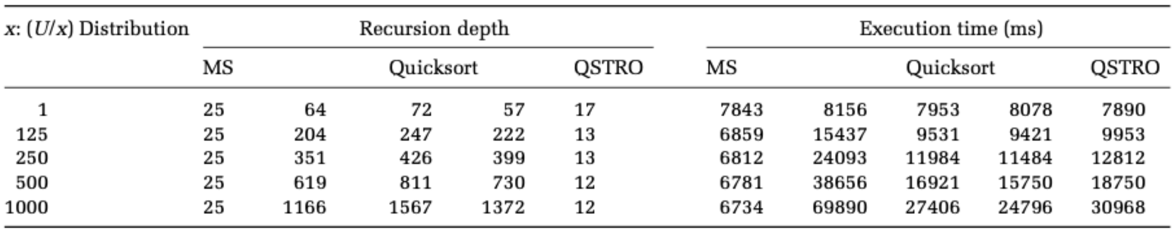
```
void Matesort(int[] A, int lo, int hi, int bitloc)
{ // initial call: bitloc = highest bit position (starting from 0)
    if ((lo < hi ) && (bitloc >=0))
    {   int k = BitPartition(A,lo,hi,bitloc);
        Matesort(A,lo,k,bitloc-1);
        Matesort(A,k+1,hi,bitloc-1);
    }
}

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].
Compared to Quicksort and Heapsort, Quicksort and Matesort run neck-and-neck and clearly outperform other algorithms. To deal with the two problems of Quicksort which cause performance degradation which also reflected in the Table1 (1. The repeated partitioning part leads to slow execution; 2. The recursion depth and execution time increase as the data repetition factor is increased), using altering pivot selection methods and coupling Quicksort with Insertion Sort are neither worked.

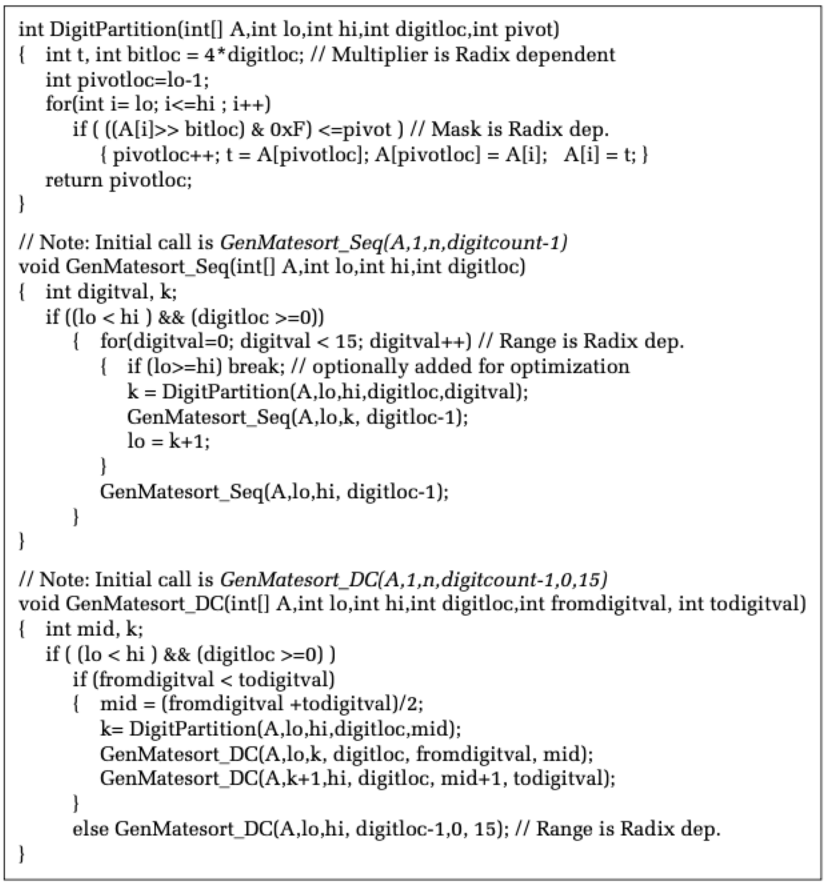


| $x$: ($U/x$) Distribution | Recursion depth | | | | | Execution time (ms) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | MS | | Quicksort | | QSTRO | MS | | Quicksort | | QSTRO |
| 1 | 25 | 64 | 72 | 57 | 17 | 7843 | 8156 | 7953 | 8078 | 7890 |
| 125 | 25 | 204 | 247 | 222 | 13 | 6859 | 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 |

Table 2 [2]

## *2.2 General radix Matesort algorithm*

The difference between General radix Matesort algorithm and basic Matesort algorithm is the former processes the data one digit at a time. When compared to binary Matesort, we can find out that Partition method has been substituted by DigitPartition method and the data need to be divided into r groups (radix r). 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(see in Listing5), one for each part, to process the data for the next digit location (digitloc – 1).



```
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.
}
```

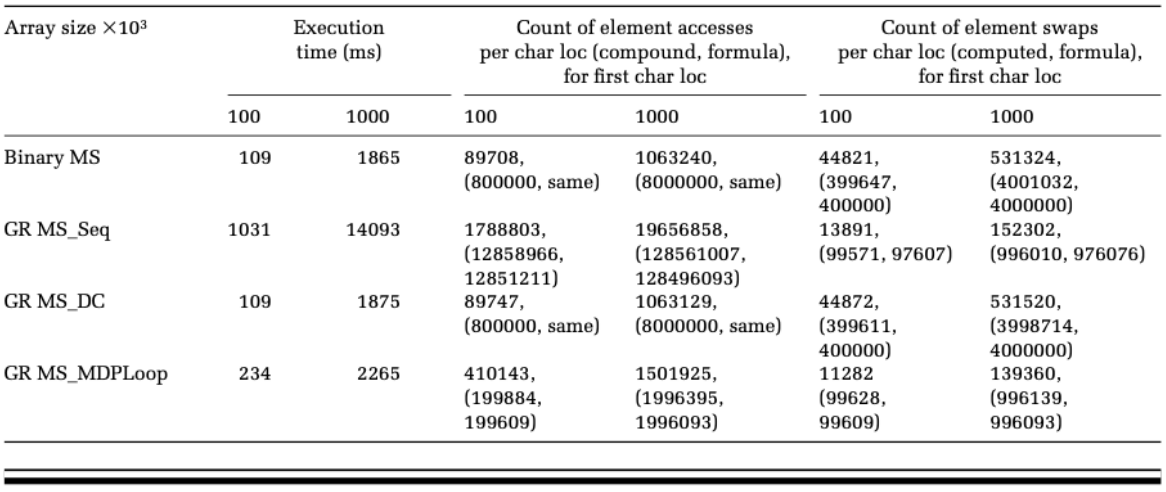
Listing 5. General radix Matesort using sequential partitioning (GenMatesort_Seq) and general radix Matesort using divide-and-conquer Partitioning (GenMatesort_DC ) – versions given are for Radix = 16[2].

For partition, a better strategy is to partition near the middle digit value. We can only consider the elements in the mid and when the data is random and uniformly distributed, the running-time result for the first digit is in $O(nlogr)$.

## *2.3 Comparison result*

When counting of element accesses and swaps (per character and for leftmost character) for various Matesort algorithms, the result shows as Table 3[2].

Table 3[2]
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.

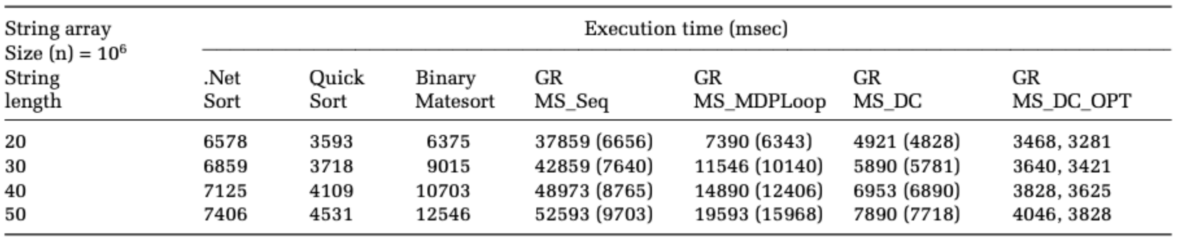


| Array size $\times 10^3$ | Execution time (ms) | | Count of element accesses per char loc (compound, formula), for first char loc | | Count of element swaps per char loc (computed, formula), for first char loc | |
|---|---|---|---|---|---|---|
| | 100 | 1000 | 100 | 1000 | 100 | 1000 |
| Binary MS | 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) |

Therefore, for random data, general radix Matesort is no better than binary Matesort.

## 3 IMPLEMENTION

General radix Matesort is able to exploit data (and encoding) redundancy. Consider several Matesort algorithms: binaryMatesort, GenMatesort_Seq, GenMatesort_MDPLoop, GenMatesort_DC and GenMatesort_DC_Opt. GenMatesort_MDPLoop, with the input data consisted of one million fixed length strings extracted from English documents [2]. From the Table 4, we can see GenMatesort_ DC performed the best timing results.



| 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 |

Table 4[2]
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.

## 4 CONCLUSION

As the data redundancy goes up, Quicksort will become slower and degenerate into the complexity of $O(n^2)$ algorithm but binary Matesort remains $O(kn)$(k: the element size in bits). And in sorting English text, the general radix Matesort is the fastest.

## 5 REFERENCES

[1] Sedgewick, Robert, Wayne, Kevin 2011. Algorithms (4th ed.). Addison-Wesley Professional. ISBN 978-0-321-57351-3. https://algs4.cs.princeton.edu
[2] Nasir Al-Darwish. 2005. Formulation and analysis of in-place MSD radix sort algorithms.

Volume: 31 issue: 6, page(s): 467-481. https://doi.org/10.1177/0165551505057007