Enhanced Generic Key-Address Mapping Sort Algorithm

Chenn-Jung Huang#, Chih-Tai Guan and Yi-Ta Chuang
Institute of Learning Technology, College of Science
National Hualien University of Education, Hualien, Taiwan
cjhuang@mail.nhlue.edu.tw

Abstract

Various methods, such as address-calculation sort, distribution counting sort, radix sort, and bucket sort, adopt the values being sorted to improve sorting efficiency, but require extra storage space. This work presents a specific key-address mapping sort implementation. The proposed algorithm has the advantages of linear average-time performance and no requirement for linked-list data structures, and can avoid the tedious second round of sorting required by other content-based sorting algorithms, such as Groupsort. The key-address mapping function employed in the proposed algorithm can fit data in any specific distribution when the mapping function is carefully designed. The cases for the uniformly and normally distributed data are explored herein to demonstrate the effectiveness of the proposed key-address mapping functions. Although the computation of the average and the standard deviation increases the overhead in our sorting algorithm, the empirical results indicate that the proposed sorting algorithm is still faster than both Quicksort and Groupsort for lists comprising 1,000 to 2,000,000 positive integers. The proposed algorithm adopts a valid key-address mapping function for uniformly distributed data, and a desirable approximation of the cumulative distribution function by using a cubic polynomial for normally distributed data, respectively.

1 Introduction

Sorting is an extensively studied problem in computer science for various practical and theoretical reasons. Sorting constitutes a fundamental operation of many computing tasks, including VLSI design, digital signal processing, network communications, database management and data processing, for which sorting operations are estimated to account for over 25% of the total processing time. The significance of sorting is reflected in the multitude of many sorting algorithms that have been presented in recent decades. Sorting a generic list of numbers is a well-studied problem that can be efficiently solved with using generic algorithms, such as Quicksort [1-2], Shellsort [3] and Mergesort [4].

Most versions of Quicksort [1] need $O(n^2)$ comparisons in the worst case, but are efficient in the average case. Furthermore, the number of data movements is very small in Quicksort. The average number of comparisons of the best-of-three version of Quicksort, known as Clever Quicksort, has been calculated as about $1.188(n+1)\log(n\tilde{}\ 2.255)\cdot n+2.507$ [5]. Mergesort is a sorting algorithm approaching the lower threshold of $O(n \log n)$. Mergesort can be shown to make $n \log n - (n - 1)$ comparisons in the worst case and $n \log n - 1.2645n$ comparisons in the average case [6–7], but requires $O(n)$ additional storage. The merging technique, which is a fundamental operation of Mergesort, has been examined for a long time as a way to alleviate this additional space requirement. Kronrod [8] developed a linear-time algorithm for merging two sorted lists in constant extra space using an internal buffer. Horvath [9], Mannila and Ukkonen [10], Pardo [11] and Huang & Langston [12] continued developing this approach, but most of their algorithms are cumbersome and inefficient. The fastest version of merging using constant extra space, introduced by Huang and Langston [12], requires at most around $1.5n$ comparisons and $2n$ exchanges [12]. The standard merging requires at most n comparisons and n exchanges.

An alternative way to decrease the sorting times is to modify the model for determining the key order. Most classic sorting algorithms adopt the "comparison based" model, i.e., they sort the list exclusively using pair-wise comparison. However, in "content-based" sorting methods, the content of keys is applied to obtain their position without needing to compare them to each other. This approach improves the analytical results because real machines allow many operations other than comparison [13]. Examples of content based sort algorithms include Bucketsort [14], Radixsort [13] and Groupsort [4].

Radixsort is a fast stable sorting algorithm for sorting items identified using unique keys. Every key is a string or number. Radixsort sorts these keys in a particular lexicographic-like order. The

algorithm operates in O($n \cdot k$) time, where n denotes the number of items, and k represents the average key length. This algorithm was originally used to sort punched cards in several passes. Harold H. Seward devised a RadixSort computer algorithm at MIT in 1954. Radixsort is faster than comparison sorts in many recent applications requiring very fast processor speeds and large computer memory.

Radixsort has resurfaced as an alternative to other high-performance sorting algorithms, which need O(n log n) comparisons. Such algorithms can sort with respect to orderings that are more complex than lexicographic ones, but this is of little significance in many practical applications. Groupsort [4] is an implementation of Bucketsort, which splits the unsorted list into k groups (buckets) according to the key of each element, and sorts each group with Quicksort. Groupsort has two advantages: (i) it achieves linear average-time performance with additional storage equal to a fraction of the number of elements being sorted, and (ii) it uses no linked-list data structures because it performs all sorting by arrays. However, content-based sort algorithms such as Groupsort need two sorting stages, and may not perform well when the data not uniformly distributed. This work presents an effective key-address mapping algorithm to avoid the need for two sorting stages, as required in content-based methods found in the literature. Experimental results indicate that the proposed algorithm can calculate addresses of the data from their key values, thus avoiding the second round sorting that is required in other sorting algorithms.

The remainder of this paper is organized as follows. Section 2 briefly surveys related work. A primitive key-bag mapping sort algorithm is presented in Section 3. Section 4 describes the proposed key-address mapping sort algorithm. Section 5 presents the empirical test results to support the theoretical expectation obtained in Section 4. Conclusions are drawn in Section 6.

2 Related Work

Various sorting algorithms have been designed to employ the values being sorted to increase efficiency, but require extra storage space. One of the first such approaches is the address-calculation sort proposed by Isaac and Singleton [15]. To simplify the following discussion, assume that n records are to be sorted in increasing order according to keys stored as positive integers in the array $x[1], \ldots\ldots, x[n]$. A sorting function is needed to associates each key with an integer corresponding to an approximate location of its record in the final sorted array. For instance, given n keys with values ranging from u to v, the linear sorting function $mx + b$ can be created through the two points (u, 1) and (v, n). Then, for a given value of a key k with $u \leq k \leq v$, the approximate location of that key in the sorted list is given by $f(k) = mk + b$. Each element $x[i]$ must then be moved to its corresponding location in an output array, say, y, by setting $y[f(x[i])] = x[i]$. Unfortunately, f may not be one-to-one, so two different keys, $x[i]$ and $x[j]$, can produce $f(x[i]) = f(x[j])$.

Another example of an address-calculation sort is the Franzisort, presented by Suraweera and Al-Anzy in [16], in which the record keys range from u to v. Given that the values of the keys are unique, by using an additional array, say, $y[u..v]$, each record $x[i]$ is moved directly from its original location to its correct location in y by setting $y[x[i]]$ to $x[i]$. The sorted list is then derived by moving the nonzero elements of y sequentially back to the next available position of the original array x.

Seward [17] and Feurzig [18] independently developed a similar method for handling repeated key values based on *distribution-counting sort*, which was described succinctly by Knuth [13]. This procedure stipulates the original array x of n elements, an output array z of size n for the sorted records and an additional array of size $v - u + 1$ elements to monitor the number of times that each value of a key occurs. Unfortunately, the user has no control over the values of u and v since they are based on the given keys. Consequently, the amount of storage space used can be prohibitive. Another address-calculation sorting algorithm is Bucketsort [14], which partitions the range of the numbers to be sorted into K subranges. Each number in the list is then placed into one of K corresponding groups called buckets based on its subrange. The numbers in each bucket are recorded in a linked list to permit an indeterminate number of values. The sorting algorithms described by Knuth [13] and Cormen et al. [19] sort the numbers in the buckets are sorted with insertion sort. The final sorted list is derived by moving the numbers from each bucket to the output list in order. The advantage of bucket sort is that it moves each number quickly to its approximate location by placing it in the correct bucket. Under the assumption that the values being sorted are uniformly distributed throughout the range, the average number of elements in each bucket is approximately n/K, which can then be sorted efficiently by insertion sort. The disadvantages of this algorithm include: (i) the use of linked-list data structures, (ii) the need for extra storage for the n linked-list elements in the buckets together with K pointers to the head of each list, and (iii) the fact that insertion sort can

consume large amounts of time with many elements in a bucket. This last disadvantage can be mitigated by using the Floyd's tree sort, [20] which is more efficient than insertion sort.

Another bucket sort variant is Groupsort [4], which splits the unsorted list into k groups based on the key of each element, and sorts each group with Quicksort. Consider an example of an unsorted list as illustrated in Fig 1, with a data range 28075 (32449 – 4374 = 28075). Figure 2 shows the range of values for each group when Groupsort is performed.

Values	21211	4374	23291	16420	17849	27399	22353	29261	31970	32449
Subscripts	1	2	3	4	5	6	7	8	9	10

Fig. 1. The original array to be sorted.

Group	Range of Value
0	4374 ~ 11393
1	11394 ~ 18412
2	18413~ 25431
3	25432~ 28075

Fig. 2. The range of values for each group.

Figure 3 illustrates the marking for the starting location of each group, and Fig. 4 shows each element moving to its group moves according to the group intervals. Figure 5 shows the sorting within each group of data.

Values	21211	4374	23291	16420	17849	27399	22353	29261	31970	32449
Subscripts	1	2	3	4	5	6	7	8	9	10
Group	0	1		2			3			

Fig. 3. The starting location of each group.

Values	4374	16420	17849	21211	23291	22353	27399	29261	31970	32449
Subscripts	1	2	3	4	5	6	7	8	9	10
Group	0	1		2			3			

Fig. 4.The lists after all elements are moved to their group.

Values	4374	16420	17849	21211	22353	23291	27399	29261	31970	32449
Subscripts	1	2	3	4	5	6	7	8	9	10
Group	0	1		2			3			

Fig. 5. The list after the second stage of sorting.

3 A Primitive Key-Bag Mapping Sort Algorithm

The key-bag mapping sort presented in this Section 1 assumes a fixed number of bags to accommodate n unsorted elements, and employs a statistical mapping function that associates each key to a corresponding bag index for each element after the mapping function is computed. The statistics mapping function is defined as,

$$index = \mathrm{map}(key) = (key + bias) \cdot ratio \,, \qquad (1)$$

where key is the key value for each element; $ratio$ denotes the ratio of the linear transformation, and $bias$ represents the bias of the original key value. The $ratio$ is given by,

$$ratio = \frac{ccnt}{2\sqrt{3} \cdot STD}\,, \qquad (2)$$

where $ccnt$ represents the counts of the bags, and STD is the standard deviation of the elements. The denominator in Eq. (2) represents the range of unsorted elements, which is assumed to be within $AVG - \sqrt{3} \cdot STD$ and $AVG + \sqrt{3} \cdot STD$, where AVG is the calculated average of all unsorted elements. Notably, the ratio given in Eq. (2) is used to compute the index for each unsorted element within the effective range, i.e. [0,…,$ccnt$-1]. The following proves why $\sqrt{3}$ occurs in Eq. (2).

Suppose that a set contains n evenly distributed elements as follows,

$$a, a+d, \cdots, a+(n-1)d\,. \qquad (3)$$

The average and standard deviation can then be expressed respectively as,

$$AVG = a + \frac{n-1}{2} d\,, \qquad (4)$$

and

$$STD = \sqrt{\frac{\sum_{k=0}^{n-1}(a+kd)^2}{n} - (a + \frac{n-1}{2}d)^2} = \frac{d}{2\sqrt{3}}\sqrt{n^2-1}\,. \qquad (5)$$

Therefore, the ratio of the distribution length of the n elements to the standard deviation is given by

$$dts = \frac{(n-1)d}{\frac{d}{2\sqrt{3}}\sqrt{n^2-1}} = 2\sqrt{3}\frac{n-1}{\sqrt{n^2-1}}\,. \qquad (6)$$

When n is sufficiently large, $\lim_{n \to \infty} \frac{n-1}{\sqrt{n^2-1}} = 1$.

Thus Eq. (6) can be further simplified as,

$$dts = \lim_{n \to \infty} 2\sqrt{3}\frac{n-1}{\sqrt{n^2-1}} = 2\sqrt{3}\,. \qquad (7)$$

Meanwhile, since the indices for the elements inside each bag begin from 0, a bias constant, $bias$, is required in Eq. (1) as follows,

$$bias = -\left(AVG - \sqrt{3} \cdot STD\right) = \sqrt{3} \cdot STD - AVG\,, \qquad (8)$$

where AVG and STD denote the average and the standard deviation of the elements, respectively.

In case the computed mapping function value is out of the range $[0,\ldots,ccnt-1]$, where $ccnt$ is the counts of the bags, then the index for the corresponding bag for the unsorted elements should be corrected as,

$$index = \begin{cases} 0 & \text{if } index < -0.5 \\ ccnt-1 & \text{if } index \geq ccnt-0.5 \\ \mathrm{round}(index) & \text{else} \end{cases}, \quad (9)$$

where $\mathrm{round}(\cdot)$ is the rounding-off function.

After the elements are placed in the corresponding bags, the numbers in each bag are sorted with Quicksort.

3.1 Time and Space Complexity Analysis

Each bag is assumed to accommodate cs elements, and the counts of the bags, $ccnt$, are given by

$$ccnt = \frac{n}{cs}, \quad (10)$$

where n denotes the number of unsorted elements.

The elements inside each bag are assumed to be sorted with Quicksort, and then the total time needed for sorting n elements in $ccnt$ bags is given by:

$$ttime = ccnt \cdot (cs \cdot \log cs) = n \times \log cs. \quad (11)$$

When the unsorted numbers are not evenly distributed, cs is set to $\sqrt{n}$, and we obtain

$$ttime = n \cdot \log \sqrt{cs} = \frac{1}{2} n \cdot \log n \cdot \quad (12)$$

Accordingly, the time complexity and space complexity of the proposed algorithm are $O(n\log n)$ and $O(\sqrt{n})$, respectively.

Conversely, $cs = \log n$ is set if the unsorted numbers are evenly distributed, and

$$ttime = n \cdot \log\log n. \quad (13)$$

Thus, the time complexity and space complexity of the proposed algorithm in this case are $O(n \log \log n)$ and $O\left(\frac{n}{\log n}\right)$, respectively.

Notably, Eq. (1) does not affect the computation of $O(ttime)$ here, because its calculation of Eq. (1) only requires $O(n)$.

4 The Proposed Key-Address Mapping Sort Algorithm

In Groupsort and the primitive key-bag mapping sort introduced in Section 3, the bottleneck occurs in the second stage of the whole process, which is the Quicksort executed in each individual group/bag. Since the cost of memory is falling, $2n$ free space was allocated in this work, and the statistics mapping function was modified as in Eq. (1) to map each unsorted number into the "appropriate" position within the $2n$ free space and avoid entering the expensive second processing stage.

$$index = map(key) = D(key) \times fs + offset, \quad (14)$$

where $D(key)$ is the cumulative distribution function for each specific model, and fs and $offset$ denote the size and offset of the free space, respectively.

It is very common that the cumulative distribution function appears in an integral form for data of different distributions and the calculation for the cumulative distribution function is time consuming. We thereby try to employ a polynomial in this work to approximate the true value of the cumulative distribution function in any specific distribution. The general form of the polynomial of degree n that approaches cumulative distribution function can be expressed as,

$$\hat{D}(key) = \sum_{i=0}^{n} a_i key^i, \quad (15)$$

Here n is varied for different distributions.

The following subsections investigate two frequently adopted statistic distributions, uniform and normal distribution, to verify the effectiveness of the proposed sort algorithm.

4.1 Key-Address Mapping Function for Uniformly Distributed Data

Equation (14) is employed to map unsorted numbers into $2n$ units of contiguously free space by linear transformation. The following relationship is assumed once the unsorted numbers are evenly distributed:

$$ratio = \frac{index - fm}{key - AVG} = \frac{fs}{2\sqrt{3} \cdot STD}, \quad (16)$$

where $2\sqrt{3} \cdot STD$ denotes the range of the unsorted elements as mentioned after the appearance of Eq. (2). The linear transformation ratio given in Eq. (16) expresses the ratio required to map the range of n evenly distributed numbers into $2n$ of contiguous free space.

The statistics mapping function for uniformly distributed data can then be expressed as,

$$index = map(key) = ratio \cdot key, \quad (17)$$

where $ratio$ is given by Eq. (16).

If two or more numbers collide in the same position in the $2n$ free space, the interpolation sort

is adopted to place one colliding element into the nearest free position around the collision location. However, the possibility of the collision should be very small compared to the amount of the data if an appropriate key-address mapping function is chosen.

The following example demonstrates how the proposed algorithm works. Assume that the original list is

21211, 4374, 23291, 16420, 17849, 27399, 22353, 29261, 31970, 32449.

After the computation of Eq. (1), the index for each number becomes:

4.03, −1.40, 4.70, 2.49, 2.95, 6.03, 4.40, 6.63, 7.51, 7.66

According to Eq. (9), since index for 4374 is out-of-bounds (−1.40 < 0), the index must be adjusted to 0. Since each element is placed into its appropriate location, no extra sorting work is required, so the overhead of the second round of sorting is avoided.

4.2 Key-Address Mapping Function for Normally Distributed Data

The probability function of a variable x, normally distributed with mean μ and standard deviation σ, can be expressed as:

$$p(x)=\frac{1}{\sigma\sqrt{2\pi}}e^{-(x-\mu)^2/(2\sigma^2)}. \tag{18}$$

The cumulative distribution function is:

$$D(x)=\int_{-\infty}^{x}p(x')dx'. \tag{19}$$

Each number's index can be calculated using the mapping function as shown in Eq. (14) when its cumulative distribution function for each number has been derived. However, as mentioned earlier, the cumulative distribution function is in an integral form and is computationally expensive. Therefore, a cubic polynomial, $g(x)=a+bx+cx^2+ex^3$, was proposed in this work to approximate the original cumulative distribution function, as shown in Eq. (19).

The derivation of the coefficients in the cubic polynomial, $g(x)=a+bx+cx^2+ex^3$, can be completed by the following error function:

$$E(x)=\int_{-\infty}^{\infty}\left(a+bx+cx^2+ex^3-D(x)\right)^2p(x)dx, \tag{20}$$

Let

$$I_n=\int_{-\infty}^{\infty}x^n p(x)dx=\frac{1}{\sigma\sqrt{2\pi}}\int_{-\infty}^{\infty}x^n e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{21}$$

and

$$J_n=\int_{-\infty}^{\infty}x^n p(x)D(x)dx=\frac{1}{2\pi\sigma^2}\int_{-\infty}^{\infty}x^n e^{-\frac{(x-\mu)^2}{2\sigma^2}}\int_{-\infty}^{x}e^{-\frac{(t-\mu)^2}{2\sigma^2}}dtdx \tag{22}$$

Then Eq. (20) can be further transformed into a linear matrix:

$$\begin{bmatrix}a\\b\\c\\e\end{bmatrix}=\left[\begin{array}{cccc|c}I_0&I_1&I_2&I_3&J_0\\I_1&I_2&I_3&I_4&J_1\\I_2&I_3&I_4&I_5&J_2\\I_3&I_4&I_5&I_6&J_3\end{array}\right], \tag{23}$$

where the values of I_0, I_1, I_2, I_3, I_4, I_5 and I_6 can be derived by Eq. (21), and J_0, J_1, J_2 and J_3 by Eq. (22).

Since it is tedious to derive the coefficients of the cubic polynomial using Eq. (23), we set $z=\frac{x-\mu}{\sigma}$ and rewrite Eqs. (18) and (19) as follows to simplify the calculation:

$$p(z)=\frac{1}{\sqrt{2\pi}}e^{z^2/2}, \tag{24}$$

$$D(z)=\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{z}e^{z'^2/2}dz', \tag{25}$$

and the cubic polynomial which approximates the cumulative distribution function becomes:

$$g(z)=a+bz+cz^2+ez^3, \tag{26}$$

After the substitution for z in Eq. (26) using $\frac{x-\mu}{\sigma}$, we obtain

$$g(x)=a+b\frac{x-\mu}{\sigma}+c\left(\frac{x-\mu}{\sigma}\right)^2+e\left(\frac{x-\mu}{\sigma}\right)^3, \tag{27}$$

and the error function becomes:

$$E=\int_{-\infty}^{\infty}\left(a+bz+cz^2+ez^3-D(z)\right)^2p(z)dz, \tag{28}$$

Since $D(z)$ is symmetric at $(0,\frac{1}{2})$, it can be

$$a=\frac{1}{2}, \tag{29}$$

and

$$c=0. \tag{30}$$

Meanwhile, the error function as given in Eq. (20) is expressed as,

$$E=\int_{-\infty}^{\infty}\left(\frac{1}{2}+bz+ez^3-D(z)\right)^2p(z)dz. \tag{31}$$

Notably, the difference between the cubic polynomial and the cumulative distribution

function becomes zero when appropriate values are chosen for b and e in Eq. (31). That is,

$$\frac{\partial E}{\partial b} = 0 = \frac{\partial E}{\partial e} \cdot \quad (32)$$

Based on Eq. (32), it can be shown that:

$$\int_{-\infty}^{\infty}\left(\frac{1}{2}+bz+ez^3\text{-}D(z)\right)zp(z)dz = 0, \quad (33)$$

and

$$\int_{-\infty}^{\infty}\left(\frac{1}{2}+bz+ez^3\text{-}D(z)\right)z^3 p(z)dz = 0 \cdot \quad (34)$$

Now Eq. (23) can be rewritten as,

$$\begin{bmatrix} b \\ e \end{bmatrix} = \left[\begin{array}{cc|c} I_2 & I_4 & J_1\text{-}\frac{1}{2}I_1 \\ I_4 & I_6 & J_3\text{-}\frac{1}{2}I_3 \end{array}\right], \quad (35)$$

where

$$I_n = \int_{-\infty}^{\infty} z^n p(z)dz, \quad (36)$$

and

$$J_n = \int_{-\infty}^{\infty} z^n p(z)D(z)dz. \quad (37)$$

We next derive the value of I_n and J_n.
Since

$$(-z)^n p(-z) = -z^n p(z), \quad (38)$$

when n is odd, thus

$$I_{2n\text{-}1} = 0, \forall n \in N, \quad (39)$$

when n is even, I_n becomes a recurrence relation as follows,

$$I_{n+2} = (n+1)I_n. \quad (40)$$

Thus, we obtain

$$I_2 = 1\ ;\ I_4 = 3\ ;\ I_6 = 15. \quad (41)$$

Next we let

$$K_n = \int_{-\infty}^{\infty} z^n p^2(z)dz. \quad (42)$$

It can be shown that:

$$J_{n+1} = nJ_{n\text{-}1} + K_n, \quad (43)$$

and we have

$$J_1 = \frac{1}{2\sqrt{\pi}}\ ;\ J_3 = \frac{5}{4\sqrt{\pi}} \cdot \quad (44)$$

Accordingly, Eq. (35) can be expressed as:

$$\begin{bmatrix} b \\ e \end{bmatrix} = \left[\begin{array}{cc|c} 1 & 3 & \frac{1}{2\sqrt{\pi}} \\ 3 & 15 & \frac{1}{4\sqrt{\pi}} \end{array}\right] = \frac{1}{24\sqrt{\pi}}\begin{bmatrix} 15 \\ \text{-}1 \end{bmatrix}, \quad (45)$$

and the cubic polynomial as given by Eq. (27) becomes:

$$g(x) = \frac{1}{2} + \frac{5}{8\sqrt{\pi}}\left(\frac{x-\mu}{\sigma}\right) - \frac{1}{24\sqrt{\pi}}\left(\frac{x-\mu}{\sigma}\right)^3. \quad (46)$$

Then the index for each number can be obtained from the following mapping function:

$$index(x) = g(x) \cdot fs, \quad (47)$$

where fs denotes the size of the free space.

4.3 Time and Space Complexity Analysis

4.3.1 Time and Space Complexity Analysis for Uniformly Distributed Data

To sort the uniformly distributed data, the proposed sort algorithm can be divided into three processing stages, which are the computation of the value of *ratio* as given by Eq. (16), the mapping of each element's key onto its corresponding address, and moving the sorted elements into the original array, respectively. The total computation time of the sorting can then be expressed as

$$T_{total}(n) = T_1(n) + T_2(n) + T_3(n), \quad (48)$$

where $T_i(n)$ denotes the computation time at the ith processing stage.

The computation of *ratio* as given by Eq. (16) requires n additions, n+1 multiplications and one square-root operation for the derivation of the standard deviation, and n-1 additions and one multiplication for the calculation of the mean, respectively. Thus, the total computation time spent at the first stage is,

$$T_1(n) = (2n-1)T_{Add} + (n+4)T_{Mul} + T_{Sqrt}, \quad (49)$$

where T_{Add}, T_{Mul}, and T_{Sqrt} denote the computation time of a single addition, multiplication, and square-root operation, respectively.

During the second processing stage, the mapping of each element's key onto its corresponding address requires one multiplication,

When the sorted elements are moved back to the original array during the final stage,
it involves n movements:

$$T_3(n) = n \cdot T_{Mov}, \quad (51)$$

where T_{Mov} denote the computation time of a single movement operation.

Accordingly, the total computation time required for sorting the uniformly distributed data becomes,

$$T_{total}(n) = ((2n-1)T_{Add} + (n+4)T_{Mul} + T_{Sqrt}) + (T_{Mul}) + (n \cdot T_{Mov})$$
$$= (2T_{Add} + T_{Mul} + T_{Mov})n + (5T_{Mul} + T_{Sqrt} - T_{Add}) \cdot \quad (52)$$

Apparently, the time complexity of Eq. (52) is $O(n)$.

Meanwhile, the space required for sorting the uniformly distributed data is $2n$ as mentioned in Section 4.1, the space complexity is $O(n)$ as well.

4.3.2 Time and Space Complexity Analysis for Normally Distributed Data

As for the normally distributed data, the whole sorting algorithm can be also separated into three processing stages, which are the calculation of mean and standard deviation, the mapping of each element's key and its corresponding address, and moving the sorted elements back to the original array, respectively.

During the first stage, the calculation of mean and standard deviation requires,

$$T_1(n) = (2n-1)T_{Add} + (n+2)T_{Mul} + T_{Sqrt}. \quad (53)$$

Then we check if there is more than one element mapped to the same address during the second processing stage, which is the mapping of each element's key and its address. The possibility of conflicting at the same address should be effectively reduced if the avoidance of the second round sorting is expected. One comparison is required if there are two elements mapped to the same address. In case there are more than two elements mapped to the same address, Quicksort is employed in this work to sort these conflicted elements.

10,000 sets of normally distributed data truncated in the range 0~1,500,000, with mean μ = 750,000 were generated using a normal random variables generator with list size equal to 1,000,000 elements to investigate the distribution of different elements which are mapped into identical addresses when the proposed sorting algorithm is used to sort the normally distributed data. Table 1 lists the statistics of the distribution of the number of the elements mapped onto the same address after we ran the sorting tests for 10,000 times using the normally distributed data. The average occurrence frequency and the standard deviation of the occurrence frequency for various numbers of conflicted elements are given at the second and the fifth columns in Table 1, respectively. The small standard deviation values comparing with the average occurrence frequency as listed in Table 1 exhibit that the behavior of the proposed key-address mapping function is indeed stable and consistent, and it is considered to be reliable to apply the statistics given in Table 1 in the computation of the time complexity for the second processing stage of the proposed algorithm.

Table 1: Distribution of number of elements mapped into the same addresses

Number of elements mapped onto the same address	**Average occurrence frequency**	**Minimum occurrence frequency**	**Maximum occurrence frequency**	**Standard deviation**
0	396160.00	394992	397269	319.25
1	369370.00	367472	371110	491.84
2	179100.00	177824	180416	336.32
3	59916.00	59016	60651	206.15
4	15485.00	15044	15901	114.98
5	3279.90	3065	3478	54.823
6	590.42	506	673	24.019
7	92.56	61	130	9.7117
8	12.92	2	27	3.5671
9	1.64	0	8	1.2794
10	0.19	0	4	0.43877
11	0.02	0	2	0.14898
12	0.00	0	1	0.053774

The maximum occurrence frequency of various numbers of conflicted elements as given at the fourth columns in Table 1 is used to compute the upper bound of the time complexity of the second processing stage,

$$\mathrm{T}_2(n)=\frac{180416}{1000000}n+\frac{60651}{1000000}n\cdot 3^2+\frac{15901}{1000000}n\cdot 4^2+\frac{3478}{1000000}n\cdot 5^2$$

$$+\frac{673}{1000000}n\cdot 6^2+\frac{130}{1000000}n\cdot 7^2+\frac{27}{1000000}n\cdot 8^2+\frac{8}{1000000}n\cdot 9^2$$

$$+\frac{4}{1000000}n\cdot 10^2+\frac{2}{1000000}n\cdot 11^2+\frac{1}{1000000}n\cdot 12^2$$

$$=1.101401\cdot n, \quad (54)$$

where i^2 represents the upper bound of sorting time when Quicksort is used to sort i numbers.

During the final stage, the sorted elements are moved back to the original array, and it takes n movement operations,

$$\mathrm{T}_3(n)=n\cdot T_{Mov}. \quad (55)$$

The total computation time for the key-address mapping sort algorithm used for normally distributed data is then

$$\mathrm{T}_{total}(n)=\left((2n-1)T_{Add}+(n+2)T_{Mul}+T_{Sqrt}\right)+(1.101401\cdot n)+(n\cdot T_{Mov})$$

$$=(2T_{Add}+T_{Mul}+1.101401)n+(-T_{Add}+2T_{Mul}+T_{Sqrt}). \quad (56)$$

Thus the time complexity of the key-address mapping algorithm is also $\mathrm{O}(n)$.

The space complexity of the proposed algorithm can be derived as follows. We first need a fixed space, which is equal to 1.5n, to save for the outcomes of the key-address mapping function. In case there is more than one element mapped to the same address, a dynamic allocation of memory is required to save the conflicted numbers. Base on the statistics as given in Table 1, the upper bound of the dynamically allocated memory required for the conflicted numbers can be expressed as,

$$asize(n)=\frac{180416}{1000000}n+\frac{60651}{1000000}n+\frac{15901}{1000000}n+\frac{3478}{1000000}n$$

$$+\frac{673}{1000000}n+\frac{130}{1000000}n+\frac{27}{1000000}n+\frac{8}{1000000}n$$

$$+\frac{4}{1000000}n+\frac{2}{1000000}n+\frac{1}{1000000}n$$

$$=0.261291n. \quad (57)$$

The space complexity of the proposed algorithm is thus equal to

$$S(n)=1.5n+asize(n)=1.5n+0.261291n=1.761291n, \quad (58)$$

which is also $\mathrm{O}(n)$.

5 Performance Evaluation

A series of simulations was conducted to evaluate the performance of the proposed sorting algorithms and contrast their results with those of Quicksort and Groupsort. The tests are performed on a personal computer with an INTEL Pentium 4 CPU. The proposed sort algorithm and Groupsort code were written in Dev-C++. The C function *sort* was used for the implementation of Quicksort.

The performance of the proposed algorithm was first evaluated using 30 uniformly distributed data sets generated with a uniform random variables generator, for list sizes varying from 1,000 to 1,600,000. Figure 6 shows that the average sorting time for each scheme is similar when the number of data in the set is small. The sorting time of the proposed algorithm improves as the size of the data set increases. The mapping function apparently reduces the probability of collision, as compared with other content-based sorting algorithms. Consequently, the proposed algorithm does not require the second round of sorting employed by other algorithms. Although the computation of the average and the standard deviation might increase the overhead of the proposed algorithm, the impact of the computation decreases significantly as the data size increases.

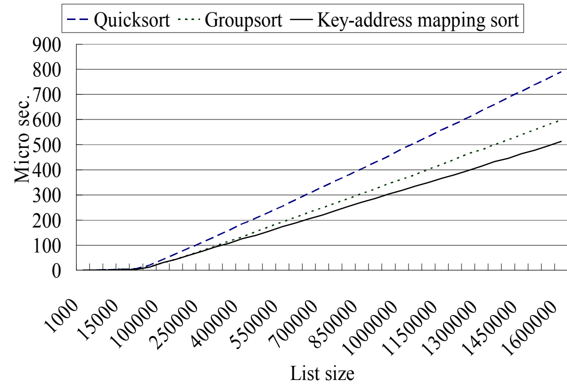

Fig. 6. Performance comparison of sorting uniformly distributed data sets.

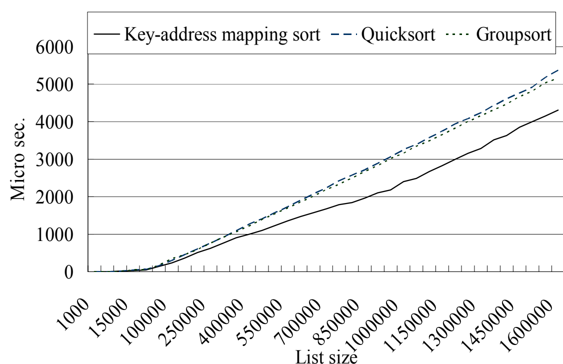

Fig. 7. Performance comparison of sorting normally distributed data sets.

The performance for a normally distributed data set was also investigated. As revealed in Fig. 7, the proposed algorithm still outperforms others, obviously owing to a suitable selection of coefficients for the cubic polynomial used in the key-address mapping function as in Eq. (46), thus reducing the possibility of dealing with colliding numbers. Meanwhile, the array elements are truncated in the range $0\sim 1.5\cdot LSize$, the average is set to $\frac{1.5\cdot LSize}{2}$, and the standard deviation is set to ten in this experiment, where *Lsize* denotes the number of the array elements. Fig. 7 exhibits that the performance of Groupsort is similar to that of Quicksort for normally distributed data when the standard deviation is not large. Based on experimental results, it is verified that the proposed algorithm indeed outperforms others whether the data is uniformly or normally distributed. Furthermore, the genetic key-address mapping function as shown in Eq. (14) was shown to be valid when an appropriate cumulative distribution function is chosen to match the specific distribution model.

In the next series of tests, we examine how the efficiency of the proposed sort algorithm is affected by a normal distribution of values, in comparison to Groupsort and Quicksort. The array elements are generated according to a normal distribution truncated in the range 0~1,500,000, with mean μ = 750,000 and standard deviation varying from 2 to 15000. Figure 8 shows the average running times comparison of the three sort algorithms with list size n = 1,000,000. It can be observed that the performances of the proposed algorithm and Quicksort are insignificantly affected by the varied standard deviation, whereas that of Groupsort is somewhat improved when the variance is increased. Meanwhile, the proposed algorithm is much faster than other two sort algorithms regardless of value of variance.

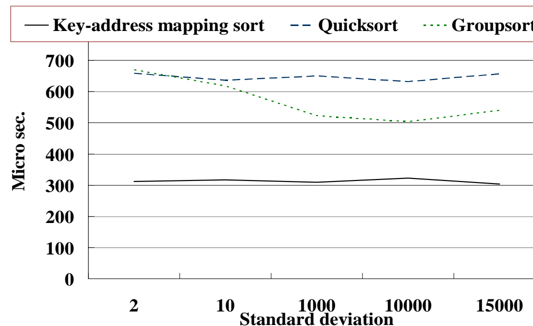

Fig. 8. Average running times for normal distribution with list size n = 1,000,000.

6 Conclusion

This work presents a key-address mapping sort algorithm to eliminate the second round of sorting required by content-based sorting algorithms found in literature. Additionally, this work demonstrates that a genetic key-address mapping function can be used to fit for the data in any specific distribution in case a fast cumulative distribution function can be obtained. Cases involving uniformly and normally distributed data were studied in this work to show the feasibility of the proposed algorithm. Experimental results verify that the proposed algorithm performs better performance than Quicksort and Groupsort with large data sets. Moreover, the running time rises rather slowly as the set size increases. Although computing the mean and standard deviation increases the computation time in the proposed sorting algorithm, approximating the cumulative distribution function for uniformly and normally distributed data using a linear equation and a cubic polynomial derived herein, respectively, can reduce the total computation time by eliminating the second round of sorting. Future work will further derive the approximate cubic polynomial for the cumulative distribution function adopted in a specific distribution model, such as Poisson or Weibull distribution to confirm the effectiveness of the genetic key-address mapping function. The applicability of the proposed work to the parallel sort computation problems will also be investigated.

7 Acknowledgement

The authors would like to thank the National Science Council of the Republic of China, Taiwan for financially supporting this research under Contract No. NSC 94-2213-E-026-001.

References

[1] C. A. R. Hoare, "Quicksort," *Comput. J.* vol. 5, no. 4, pp. 10–15, 1962.
[2] R. Sedgewick, "The analysis of quicksort programs," *Acta Informatica*, vol. 7, pp. 327-355, 1977.
[3] D. L. Shell, "A high speed sorting procedure," *Communications of ACM*, vol. 2, no. 7, pp.30–32, 1959.
[4] A. Burnetas, D. Solow, and R. Agrawal, "An analysis and implementation of an efficient in-place bucket sort," *Acta Informatica*, vol. 34, pp. 687–700, 1997.
[5] I. Wegener, "Bottom-up heap sort, a new variant of heap sort beating on average quicksort if n is not very small," in *Proceedings of Mathematical Foundations of Computer Science*, pp. 516-522, 1990.
[6] J. H. Kingston, "Algorithms and Data Structures: Design, Correctness, Analysis," pp.175-194, Addison-Wesley, Reading, MA, 1990.
[7] D. E. Knuth, "The Art of Computer Programming," Vol. 3, Addison-Wesley, Reading, MA, 1973.
[8] M. A. Kronrod, "An optimal ordering algorithm without a field of operation," *Dokl. Akad. Nauk SSSR*, vol. 186, pp. 1256-1258, 1969.
[9] E. C. Horvath, "Stable sorting in asymptotically optimal time and extra space," *J. Assoc. Comput*., pp. 177-199, 1978.
[10] H. Mannila and E. Ukkonen, "A simple linear-time algorithm for in situ merging," *Informat. Process. Lett.,* vol. 18, pp. 203-208, 1984.
[11] L. T. Pardo, "Stable sorting and merging with optimal space and time bounds," S*IAM J. Comput.*, vol. 6, pp. 351-372, 1977.
[12] B. C. Huang and M. A. Langston, "Practical in-place merging," *Comm. ACM*, vol. 31, pp. 348-352, 1988.
[13] D. Knuth, "The Art of Computer Programming," vol. 3. 1979.
[14] L. Devroye, "Lecture notes on bucket algorithms," Birkhauser, 1986.
[15] E. J. Isaac, and R. C. Singleton, "Sorting by address calculation," *J. ACM* vol. 3, pp. 169–174, 1956.
[16] F. Suraweera, and J. M. Al-Anzy, "Analysis of a modified address calculation sorting algorithm," *Comput. J.* vol.31, no.6, pp. 561–563, 1988.
[17] H. H. Seward, "Information sorting in the application of electronic digital computers to business operations," *Tech. rep.*, *MIT Digital Computer Laboratory,* Report R-232, Cambridge, Mass, 1954
[18] W. Feurzig, "Algorithm 23, math sort," *Commun. ACM*, vol. 3, pp. 601–602, 1960.
[19] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to algorithms," MIT Press, 1990.
[20] R. W. Floyd, "Algorithm 245: Treesort 3," *Commun. ACM*, vol. 7, no. 12, p. 701, 1964.
[21] E. Gamson and C. Picard, "Algorithme de tri par addressage direct," *C. R. Acad. Sc. Paris* vol. 269, pp. 38-41, 1969.
[22] F. Ducoin, "Tri par addressage direct. R.A.I.R.O.," *Informatique/Computer Science*, vol. 13, no. 3, pp. 225–237, 1979.
[23] G. H. Gonnet, "Handbook of Algorithms and Data Structures," Addison-Wesley, 1984.