Novel Hash-Based Radix Sorting Algorithm

Paul K. Mandal
Department of Computer Engineering
California State University Fullerton
Fullerton, USA
pmandal@csu.fullerton.edu

Abhishek Verma
Department of Computer Science
New Jersey City University
Jersey City, NJ 07305
averma@njcu.edu

***Abstract*— Sorting remains a quintessential problem in computer science, considerable research has focused on how to sort more efficiently a collection of elements. Although there are many algorithms that can handle the sorting of objects, most are comparison sorts. o sort objects in linear time, either Bucket Sort or Radix Sort can be used. With both algorithms, the corresponding array indices represent a hash for the object. However, Radix sort also requires an auxiliary array. In this paper, a hash table in place of the array in the Radix Sort algorithm is proposed. Using a hash table in place of the array in Radix Sort should avoid the calculations for the array and are better suited for handling objects than static arrays. As with an array-based Radix sort, the hash-based Radix Sort should maintain linearity. This methodology thereby should sorting objects efficiently and in linear time.**

Keywords— sorting; counting sort; radix sort; hash; chaining.

I. INTRODUCTION

Sorting is a process to order a set of elements by some parameter in order to handle them more efficiently [1]. For example, within a database of people, searching for someone would be facilitated greatly if the database names were sorted in alphabetical order. Sorting and searching remain two of the oldest and oft-studied problems in algorithm programming.

The amount of information found on the internet doubles every two years [2]. This exponential growth concomitantly has increased the literature on sorting algorithms. Of the well-known sorting algorithms (quick sort, insertion sort, selection sort, merge sort, and others), merge sort and quick sort are unique since they possess an average runtime of O(n log n). Merge sort's worst case runtime also is O(n log n) while quick sort's worst case runtime is $O(n^2)$ [3]. Counting sort and radix sort perform much better in terms of time efficiency. However, the arrays used in counting sort do not lend themselves well to sorting objects. Although radix sort can sort objects in linear time, it still requires the use of an auxiliary array [4]. Because of the inherent inefficiency in radix sort for objects, the following hypothesis now is proposed: replacing the array with a hash table could (1) sort objects; (2) obviate much of the arithmetic calculations required for counting sort, and (3) still run in linear time independent of the number of elements to be sorted.

This remainder of this paper is organized in the following sections: section II outlines concept of this hash sort algorithm,; section III covers how hash sort works; section IV discusses details of hardware and software specifications used; section V outlines how we tested hash sort was tested along with the results; section VI summarizes the findings; and-- finally-- section VII proposes avenues for future investigations.

II. BACKGROUND

Understanding the current literature of sorting algorithms and data structures provides insight to how the hypothesis of replacing the array with a hash table in radix sort was developed.

A. *Comparison Sort:*

The sorted order determined by a comparison sort is based solely on comparing the elements. For a worst case scenario, a comparison sort must make Ω(n lg n) comparisons to sort n elements [4,5]. Consequently, the time required to sort a greater number of elements increases exponentially. Henceforth, improving comparison sorts would only bring marginal differences of sorting time, at the very best yielding improvements by a factor of some constant [4]. In order to sort in linear time, a different sorting algorithm is required.

B. *Counting Sort:*

The simplest counting sort works in the following manner: for a set of positive integers, an array with a length equivalent to that of the largest number in the set is created. Then, loop through the set, and for every element *e*, increment the index of the array at *e* by 1. Then loop through the array, and decrement each index until it reaches zero, adding a number to the set equivalent to the value of the index each time. This process is shown in Fig. 1.

Intuitively, there are two problems with this rather simple counting sort example: (1) It is memory intensive (it must create an array that is in length as long as the highest number) and (2) It does not use memory efficiently if the number of elements in the list is not close to the largest number in the list [4].

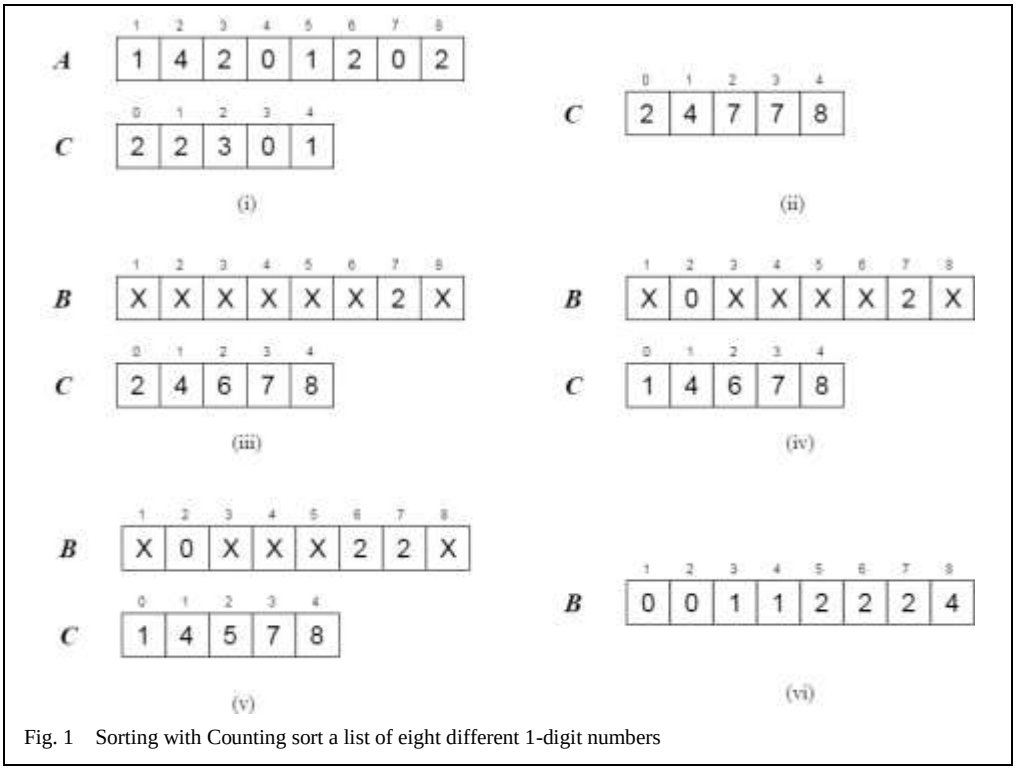

Fig. 1 Sorting with Counting sort a list of eight different 1-digit numbers

```
COUNTINGSORT(A[], B[], k)
   for I = 0 to k
       C[i] = 0
   for j = 1 to A.length
       C[A[j]] = C[A[j]] + 1
   for i = 1 to k
       C[i] = C[i] + C[i – 1]
   for j = A.length downto 1
       B[C[A[j]]] = A[j]
       C[A[j]] = C[A[j]] – 1
```

C. *Radix Sort:*

Radix sort solves the memory problem encountered in counting sort first by sorting by the least significant digit and then by identifying the next least significant digit until it reaches the most significant digit. In this case, since the digits are in base 10, the arrays are only 10 digits long. This is called Radix. If sorting uppercase letters, the Radix would be 26. Fig. 2 demonstrates this process.

Overall, radix sort is poorly written to deal with objects since counting sorts work by incrementing an index of an array. Furthermore, the stable counting sort in radix has four inner loops and a variety of arithmetic computations. Optimization for radix sort can occur in a variety of ways. Some take advantage of hardware such as CC-Radix [6]. Others add features that make it more efficient, such as using insertion sort for a small number of keys, modifying radix sort to use a 2-d array to make it more efficient [7]. Lastly, using key pointers for partitions also can augment efficiency [8]. In all instances, these optimizations do not eliminate the auxiliary array.

D. *Hash Table:*

In a hash table, the methodology for storing data requires keys, which are mapped to indices. Each element is stored in a certain index in the table given by a hash function. A hash table has a worst case search time of O(n) and an average search time of O(1) [9]. The main problem with hashing is that the number of possible indexes generated by the hash function is less than all of the potential keys. Two different keys thus can map to the same index-- commonly termed a collision.

E. *Chaining:*

Collisions can be resolved through chaining [9]. Specifically, if two elements are hashed to the same index, then one element is chained to the end of the previous element. Thus, each index in the hash can be viewed has having a corresponding list. Since new elements are inserted at the end of the hash, the elements retain the order in which they were inserted.

III. PROPOSED HASH BASED RADIX SORTING ALGORITHM

Based on the above discussion, a HASHSORT algorithm. is proposed, which is inspired from radix sort. Objects are sorted starting from the least significant digit, then the next least significant digit and so forth. The main difference is that the objects are stored in a hash rather than an array. Furthermore, if an object is being sorted by one of its parameters, the number of indices in the hash only needs to match the radix of the given parameter.

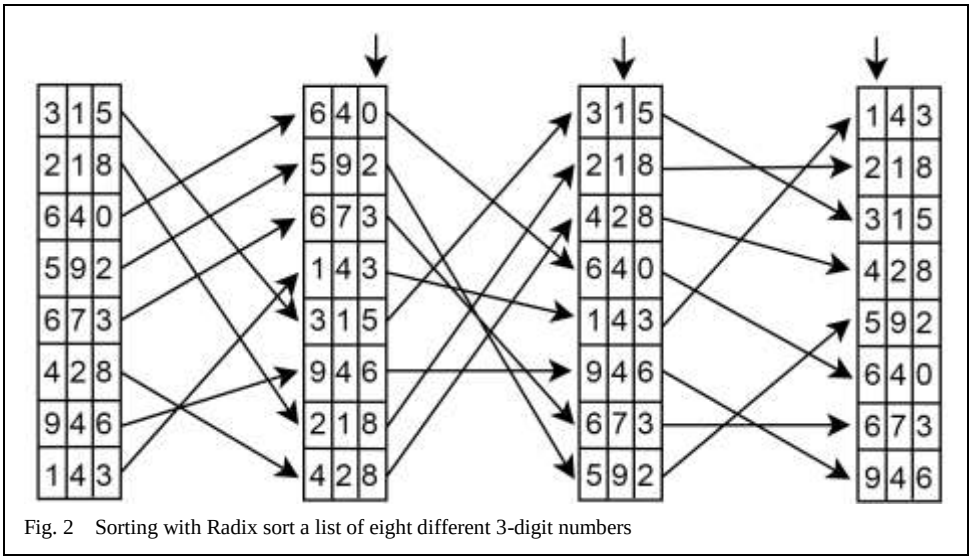

Fig. 2 Sorting with Radix sort a list of eight different 3-digit numbers

```
HASHSORT(Arr[], size, dim, Radix)
   for i = 1 to dim
       Hash h = new Hash()
       for j = 0 to size – 1
           int key = Arr[j]/Radix^(i – 1)
                     % Radix
           h.add(Arr[j], key)
       for j = 0 to size – 1
           Arr[j] = h.getFirst()
```

This method utilizes four parameters: "Arr[]" which is the sequence to be sorted, "size" which specifies the length of Arr[], "dim" which specifies the number of digits in each number, and "Radix" which specifies the radix of the numbers to be sorted. The outer loop simply traverses through each digit going from the least significant digit to the most significant digit.

A hash is created wherein the values will be temporarily stored. The first inner loop iterates through the array. A key is created that is simply the "i"-th digit of the "j"-th number. The number then is inserted into the hash at the corresponding index. The second inner loop removes each object from the hash.

Since hash sort does not run counting sort in its inner loops like a regular radix sort does, it does not require the arithmetic used in the auxiliary array. If pointers are managed correctly, more efficient sorting of objects in linear time should occur.

IV. Hardware and Software

Hash sort was tested on a Virtual Machine (VM) using VMWare Workstation. The VM had 8 gigabytes of ram, 4 cores from an Intel 6700K, and was connected by a 7200 RPM hard drive connected to the host by USB. Our VM ran Linux Mint 18.1, with the Mate GUI since it is less graphically intensive. Hash sort was written and compiled using C++.

V. Experimental Results and Discussion

As described above, this algorithm has two main parameters that affect its runtime: (1) the number of elements and (2) the number of digits in each number. In order to test how the number of elements affects the runtime, random sets of five-digit numbers were generated. Hash sort then was run on each of these sets for a total of five times. The results are summarized in Table I and Fig. 3. Next, as demonstrated in Table II and Fig.4, by fixing the number of elements to 1000, the same process was repeated in order to test how changing the number of digits affected runtime.

TABLE I. No. of Elements vs. Runtime Runtime for Proposed Hash Based Radix Sorting Algorithm

Number of Elements	Runtime (ms)
10	0.0438
20	0.038
50	0.0654
100	0.0726
200	0.1532
500	0.48
1000	0.7854
2000	1.5448
5000	4.542
10000	7.9902

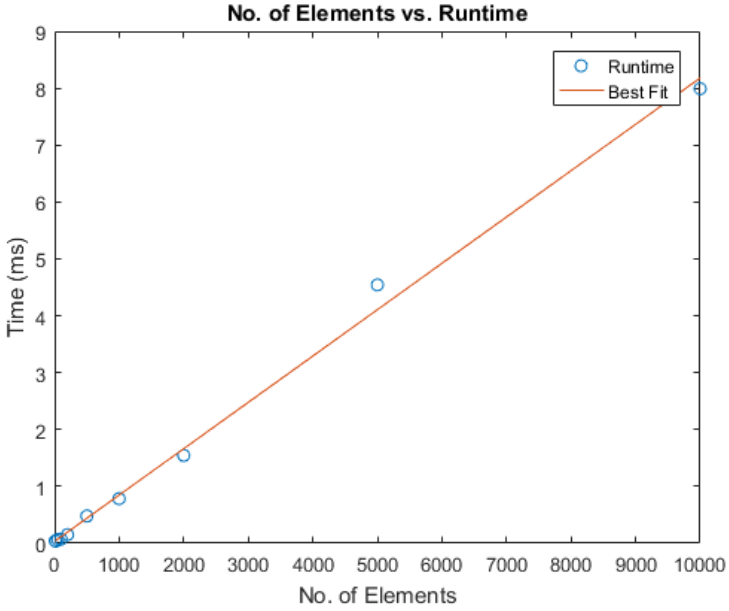

Fig. 3 Plot of number of elements vs. runtime for proposed Hash Based Radix Sorting Algorithm.

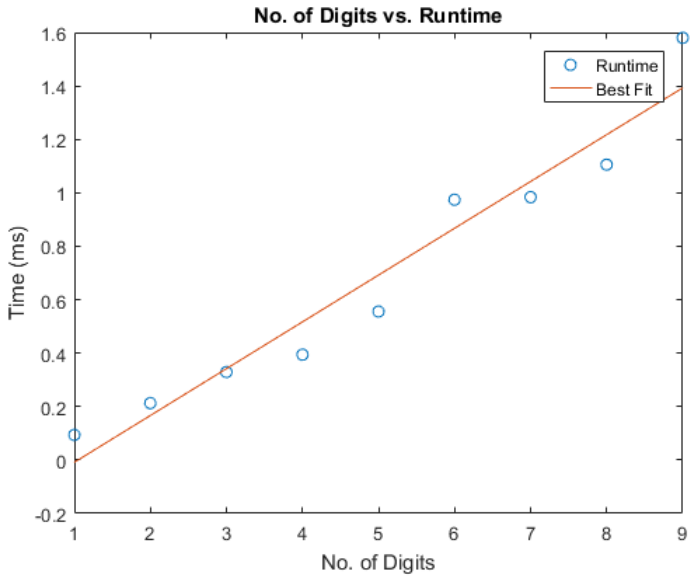

Fig. 4 Plot of number of digits vs. runtime for proposed Hash Based Radix Sorting Algorithm.

TABLE II. NO. OF DIGITS VS. RUNTIME FOR PROPOSED HASH BASED RADIX SORTING ALGORITHM

Number of Digits	Runtime (ms)
1	0.094
2	0.213
3	0.3292
4	0.3944
5	0.5558
6	0.9742
7	0.9836
8	1.1052
9	1.5798

Both Fig. 3 and Fig. 4 illustrate how hash sort scales linearly as the number of elements and digits are changed, making it O(w * n) where w denotes the number of digits and n is the number of elements. This hash data structure thus is better suited to handle objects and dynamic structures than an array. Moreover if returning an array is not required, one could return the final hash instead of a sorted array during the final step of hash sort since a hash has an O(1) search time.

VI. CONCLUSION

A new algorithm that used a hash in place of the auxiliary array in radix sort now has been proposed. A hash sort can order elements by the value of a digit in question by simply using the digit as a key. As shown, hash sort scales linearly with both the number of digits and with the number of elements.

VII. FUTURE WORK

Future work on this algorithm should focus on how to optimize traversal through the hash and calculating the keys. Leveraging advantages of hardware could also prove fruitful. Conceivably, a multithreaded version of this algorithm could be developed, thereby further decreasing the runtime., either by using the merge method in merge sort. In this manner, further gains in efficiency could be achieved by sending numbers that are closer in range to the same execution thread.

REFERENCES

[1] P.Adhikari, Review on Sorting Algorithms, "A comparative study on two sorting algorithms", Mississppi state university, 2007.

[2] Zhang, G., Zhang, G., Yang, Q., Cheng, S. and Zhou, T. (2008). Evolution of the internet and its cores. New Journal of Physics, vol. 10 no. 12, p.123027.

[3] K. Al-Kharabsheh, I. AlTurani, A. AlTurani, and N. Zanoon, "Review on sorting algorithms a comparative study," *International Journal of Computer Science and Security (IJCSS)*, vol. 7, no. 3, 2013

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Sorting in linear time" in *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2014, ch. 8, pp. 191-212.

[5] D. Knuth, *The Art of Computer Programming*, 3rd ed., vol. 3, Addison Wesley, 1973, ch 5, pp. 168 - 179

[6] D. Jimenez-Gonzalez, J. J. Navarro, and J. Larriba-Pey, "CC-Radix: a cache conscious sorting based on Radix sort," *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2003. Proceedings., Genova, Italy, 2003, pp. 101-108.

[7] P. M. McIlroy and K. Bostic, "Engineering radix sort," *Computing Systems*, vol. 6, no. 1, 1993

[8] I. J. Davis, "A fast radix sort," *The Computer Journal*, vol. 35, no. 6, December 1992 Pages 636–642.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Hash tables" in *Introduction to Algorithms, 3rd ed. Cambridge*, MA: MIT Press, 2014, ch. 11, pp. 253-285.