Performance of Efficient Sorting Algorithms for Duplicate Data

Kinjalkumar A.Prajapati
B.S.Patel Polytechnic,
Ganpat Vidyanagar, Mehsana, Gujarat
E-mail:kap_1989@yahoo.com

Abstract-*Today many comparisons based sorting algorithm that cope with popular task of sorting. Quick sort is one of divide and conquer based algorithm, which has O (n log n) complexity for n data values. This paper titled Improved Quick sort Algorithm is based on duplicate input data. Till now there are many papers on the random data values. But none of them identify the issue that what about the duplicate data which are used at different applications. To have some experimental data to sustain these comparisons three representative algorithms were chosen (classical quicksort, merge sort, heapsort). The improved quicksort gives better average time and no of comparisons for repeated data***.

Keywords- ***Algorithm, Comparisons, Quick sort, Duplicate data, Sorting***

I.INTRODUCTION

Sorting is a fundamental task that is performed by most computers. It is used frequently in a large variety of important applications. All spreadsheet programs contain some sort of sorting code. Database applications used by insurance company, banks, and other institutions all contain sorting code. Because of the importance of sorting in these applications, many sorting algorithms have been developed with varying complexity. Bubble sort, insertion sort, and selection sort are slow sorting algorithms have a theoretical complexity of O (N2). Even though these algorithms are very slow for sorting large arrays, the algorithm is simple, so they are not useless.

If an application only needs to sort small arrays, then it is satisfactory to use one of the simple slow sorting algorithms as opposed to a faster, but more complicated sorting algorithm.

For these applications, the increase in coding time and probability of coding mistake in using the faster sorting algorithm is not worth the speedup in execution time. Of course, if an application needs a faster sorting algorithm, there are certainly many ones available, including quick sort, merge sort, and heap sort. These algorithms have a theoretical complexity of O (N log N). They are much faster than the O(N2) algorithms and can sort large arrays in a reasonable amount of time. However, the cost of these fast sorting methods is that the algorithm is much more complex and is harder to correctly code. But the result of the more complex algorithm is an efficient sorting method capable of being used to sort very large arrays. In this paper, a comparative performance evaluation of improved Quick Sort with three different sorting algorithms:

Quick sort, Heap sort, and Merge sort with repeated data is presented. Quick sort is an in-place sorting algorithm. In-place sorting algorithms play an important role in many fields such as very large database systems, data warehouses, data mining, etc. Such algorithms maximize the size of data that can be processed in main memory without input/output operations.

II. LITERATURE SURVEY

Quick Sort is a well known fast algorithm for data sorting. It was invented by Hoare [1, 2].Quick Sort is the default sorting scheme in some operating systems, such as UNIX. Till now in the literature they considered the distinct data as given in [3-15] and there are different variations of quick sort also proposed like multikey. In none of the literature they had raised an issue when data will be repeated. In they talk about improved heap sort algorithm and compare with quick sort and existing heap sort algorithm and given test conditions on the different scenario. The duplicate data will be very useful at different places. Assume that any mobile company wants to see monthly data or yearly data based on the city or name than there are same data can be repeated many times. At railway station they displayed the list of passengers for railway in terms of chart so if the chart is based on the name of passengers then the name will be repeated often. But in all above applications the data which already sorted doesn’t give any advantage for future.

Assume that currently we search for all the mobile company customers in 2009 year and then in 2010 year. So both searches will be independent. So even insertion sort also not useful in this type of scenario. The repeated can also be useful in bank databases, Schools and other organizations. So this paper raise the issue and importance of duplicate data in various filed and according to that modified the original quick sort algorithm and get the better time and comparisons utilization. The rest of paper organize as follow: section 3 give the overview of comparison based sorting algorithms with modified quick sort algorithm, section 4 give the result analysis and section 5 will give conclusion.

III. COMPARISON BASED SORTING ALGORITHMS

In addition to varying complexity, sorting algorithms also fall into two basic categories - comparison based and no comparison based. A comparison based algorithm orders a sorting array by comparing the value of one element against the value of other elements. Algorithms such as quick sort, merge sort, heap sort, bubble sort, and insertion sort are comparison based. Alternatively, a non-comparison based algorithm sorts an array without consideration of pair wise data elements. Radix sort is a non-comparison based algorithm that treats the sorting elements as numbers represented in a base-N number system, and then works with individual digits of N.

A. Quick Sort

The quick sort is an in-place, divide-and-conquer, massively recursive sort. Quick sort is also known as a partition-exchange sort because that term captures the basic idea of the method. One of the elements is selected as the partition element. The remaining items are compared to it and a series of exchanges is performed. When the series of Exchanges is done, the original sequence has been partitioned into three subsequences:

1. All elements less than the partition element
2. The partitioning element in its final place
3. All elements greater than the partition element

At this stage, step 2 is completed and quick sort will be applied recursively to steps 1 and 3. The sequence is sorted when the recursion terminates. Quick sort runs in O (n log (n)) on the average.

B. Heap Sort

Heap sort is a comparison-based sorting algorithm. Heap sort is an in-place algorithm, but is not a stable sort. Heap sort

is an elegant and efficient sorting method based on the operation of heaps. A heap is a “complete binary tree”. It is constructed by placing a node called the root, and then going down the page from left to right, connecting the two nodes under each node until all of the nodes have been placed. The two nodes below each node are called its children, and the node above each node is called the parent. The item in each node should be larger than its children, so the root contains the largest item to be sorted. Heap sort is a true “in place” sort because it uses no extra memory and is always guaranteed to sort N elements in **N log N** steps, no matter what the inputs are. However, since its inner loop is about twice as long as that of quick sort, it is about twice as slow on average than quick sort. The basic idea is to create a heap containing the items to be sorted, and then to remove them in order. This means that an element is pulled off from the top of the heap, and then the largest element below is moved to the top. This continues until all of the elements are moved to the top and removed, yielding a sorted array.

C. Merge Sort

Merge sort is a comparison-based sorting Algorithm. It is stable, meaning that it preserves the input order of equal elements in the sorted output. It follows divide and conquer method. Merge sort runs in ***O* (*n* log (*n*))** on average. Conceptually, merge sort works as follows:

1) Divide the unsorted list into two sub list of about half the size.
2) Divide each of the two sub lists recursively until we have list sizes of length 1, in which case the list itself is returned.
3) Merge the two sub lists back into one sorted list. Elements are moved to the top and removed, yielding a sorted array.

D. Modified Merge Sort

The main disadvantage with quick sort is in worst case when data is already sorted whether in increasing or no increasing order or lexically. Even this is true for particular sub partition of a given partition. So the modification is if somehow we can check the given sub partition is already sorted than we not require selecting the pivot element and going for further procedure. The advantage of quick sort procedure is if data is already sorted in decreasing order and we assume that we always select pivot element as middle element than within one pass of the algorithm Partition, the data will be sorted in increasing order. So that’s why we use middle element as pivot element in our modified algorithm. In the original algorithm we require to change only partition algorithm and add some module in the stating of the partition algorithm which will take care of the sorted data.

```
Algorithm check_order (low, high, a [1...n])
{
i=low
while (i <high)
{
if(a[i]>a[i+1] )
break
i=i + 1
}
if ( i=high) return -1;
call Partition(low, high, a [1...n]) }
```

The above is simple call to check_order function which was called from quick sort main algorithm. And then this algorithm check whether the given algorithm is sorted or not by checking from the lower side the moment is found that the array is sorted in increasing order of its data the algorithm will return -1 or it will called the usual partition algorithm which will return the partition index in array for the pivot element.

TABLE 1
COMPARISON OF IMPROVED QUICKSORT

No. of Data	**Data Range**	**[Time][Comparisons] in µsec**			
		Merge sort	**Heap sort**	**Quick Sort**	**Modified Quick Sort**
10	1-10	[9] [101]	[3] [139]	[3] [52]	[3] [54]
100	1…100	[37] [1890]	[36] [1040]	[20] [886]	[20] [1056]
	1…10	[40] [1860]	[36] [1034]	[18] [810]	[17] [823]
1000	1…1000	[393] [28373]	[511] [15502]	[258] [13736]	[251] [14098]
	1…100	[397] [28303]	[536] [15339]	[228] [12315]	[209] [11776]
	1…10	[367] [27931]	[468] [14912]	[187] [11293]	[128] [7886]
10000	1…10000	[4927] [384471]	[6843] [297857]	[3219] [182625]	[3164] [190243]
	1…1000	[4836] [384657]	[6819] [204830]	[2961] [176192]	[2787] [175099]
	1…100	[4665] [384021]	[6565] [203929]	[2234] [139244]	[1202] [76959]
	1…10	[4358] [376509]	[6180] [198081]	[900] [136893]	[461] [70057]

IV. RESULT ANALYSIS

Table 1 shows the comparison of modified quick sort with other sorting algorithm. It shows that as repeated data grows the modified algorithm work better. The Time shown in table is in terms of Microseconds.

V. CONCLUSIONS

From the above Table1, we conclude that when repetition of data is increasing in such cases traditional merge sort, heap sort and quick sort are having larger number of comparisons.

New technique suggested reduces these comparisons by a countable margin. So, we suggest that if, we are having repeated data to sort than suggested modified algorithm would give better performance.

REFERENCES

[1] C.A.R. Hoare, *Quicksort*, Computer Journal, Vol. 5, 1, 10-15(1962).

[2]C. Hoare, FIND (Algorithm 65), Communications of ACM, 4 (1961),pp. 321–322.

[3] Knuth, D.E., 1988. *The Art of programming-Sorting and Searching*. 2nd Edn. Addison Wesley, ISBN: 020103803X.

[4] Cormen, T.H. et al. *Introduction to Algorithms*. 2nd Edn., 2001. ISBN:0262032937

[5] H. M. Mahmoud, R. Modarres, and R. T. Smythe. *Analysis of quickselect: An algo-rithm for order statistics*. ITA – Theoretical Informatics and Applications, 29(4):255–276, 1995.

[6] R. S. Francis and L. J. H. Pannan. *A parallel partition for enhanced parallel quicksort*. Parallel Computing, 18(5):543–550, 1992.

[7] H. Mahmoud, *Average-case analysis of moves in quick select*, in:Proceedings of Workshop on Analytic Algorithms and Combinatorics,ANALCO, 2009

[8] B. Vall´ee, J. Cl´ement, J. A. Fill, and P. Flajolet. *The number of symbol comparisons in quicksort and quickselect.* In 36th International Colloquium on Automata, Lan-guages and Programming (ICALP 2009),volume 5555 of Lecture Notes in Computer Science, pages 750–763,Berlin, Heidelberg, 2009. Springer.

[9] M. A. Weiss. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley,1998.[10] H. Mahmoud, R.Modarres, and R. Smythe,*Analysis of quickselect: An algorithm for order statistics*, *RAIRO*, Theoretical Informatics and Applications, 29 (1995), pp. 255–276.

[11] P. Kirschenhofer, H. Prodinger , C. Martínez, *Analysis of Hoare's FIND algorithm with median-of-three partition*, Random Structures & Algorithms, v.10 n.1-2, p.143-156, Jan.–March 1997.

[12] H. M. Mahmoud. *Average-case analysis of moves in quick select*. In C.Mart´ınez and R. Sedgewick, editors, Proc.

of the 6th Workshop on Analytic Algorithmics and Combinatorics (ANALCO), pages 35–40.SIAM, 2009.
[13] P. Hennequin, *Combinatorial analysis of Quick-sort algorithm*, RAIRO: Theoretical Informatics and Applications, 23 (1988), pp. 317–333.

.

[14] P. Kirschenhofer and H. Prodinger, *Comparisons in Hoare's Find algorithm*, Combinatorics,Probability, and Computing, 7 (1998) pp.111–120.
[15] Helmut Prodinger, *Multiple Quickselect—Hoare's Find algorithm for several elements*, Information Processing Letters, v.56 n.3,p.123-129, Nov. 10, 1995 .