MQ Sort an Innovative Algorithm using Quick Sort and Merge Sort

Renu
R N College of Engg. & Technology,
Panipat

Manisha
Assistant Professor
RN College of Engg. & Technology,
Panipat

ABSTRACT
Sorting is a commonly used operation in computer science. In addition to its main job of arranging lists or arrays in sequence, sorting is often also required to facilitate some other operation such as searching, merging and normalization or used as an intermediate operation in other operations. A sorting algorithm consists of comparison, swap, and assignment operations[1-3]. There are several simple and complex sorting algorithms that are being used in practical life as well as in computation such as Quick sort, Bubble sort, Merge sort, Bucket sort, Heap sort, Radix sort etc. But the application of these algorithms depends on the problem statement. This paper introduces MQ sort which combines the advantages of quick sort and Merge sort. The comparative analysis of performance and complexity of MQ sort is done against Quick sort and Merge sort. MQ sort significantly reduces complexity and provides better performance than Quick sort, Merge sort.

Keywords
Sorting, Merge sort, MQ sort, Bubble sort, Insertion sort, Time Complexity, Space Complexity.

1. INTRODUCTION
Algorithms have a key role in computer science. As we know that computer works on instructions and an algorithm is simply a definite sequence of instruction that we use to perform any task on computer. So informally an algorithm is a well defined procedure that is used to solve a computational problem. There are different categories of algorithms; different algorithms for different problems. Sorting algorithms are one such category. Sorting algorithms are used to arrange the given data into a logical sequence. Sorting algorithms usually work on an array or a list of elements which can be alphabets or numerals. There are several algorithms available for sorting some of which have quite simple working while others have complex working. These algorithms are problem specific means a single algorithm does work best in all situation[7,8,10].

For deciding which algorithm to use when we have to analyse the situation at hand. For example one algorithm gives optimal performance when the number of elements in the list is small while when the number of elements is large the algorithm may become unstable or may not provide optimal results or we can say its performance degrades. Likewise some algorithms work well on floating point numbers while others may have flawed performance. Other factors also contribute for selection or rejection of an algorithm for a specific problem like programming effort which increases with complexity of the algorithm, memory space available and if the data exists in main memory or an external device. These factors deduce that sorting algorithms are problem specific. But there is a direct relation between the complexity of an algorithm and its relative effectiveness [6,11].

Formally defining a sorting algorithm is a procedure that accepts a random sequence of numbers or any other data which can be arranged in a definite logical sequence as input; processes this input by rearranging these random elements in an order like ascending or descending and provides the output.

For example if the list consist of numerical data such as

Input: 22,35,19,55,12,66

Then the output would be: 12,19,22,35,55,66

12<19<22<35<55<66(given that the input had be arranged into ascending order)

OR

The output will be: 66<55<35<22<19<12

Similarly sorting algorithm can be used to order alphabetical data into a lexicographical order.

Sorting is a fundamental data structure operation in computer science. Sorting is used as intermediate step in many other operations. Sorting facilitates other operations such as searching and selecting largest or smallest item in the list. Sorting usually consist the following operations:

- Comparing two data items
- Swapping data items to get them to their correct position

For selecting a sorting algorithm for a particular problem[4] we have to consider various factors such as:

- Size of the array or list
- Memory constraints
- Condition of the input data
    - Completely sorted
    - Inversely sorted
    - Partially sorted
    - Almost sorted
    - Random

There are many categories in which we can divide the sorting algorithms. One such category is called internal or external sort depending on whether the data is stored in internal or external memory. Another way to categorize sorting algorithms are on based on their complexities. The two groups of sorting algorithms based on complexities are O(n2), which includes the bubble, insertion, selection sort and O(nlogn)

which includes the merge, heap & quick sort.

The paper is organized into following sections. In section 2, related work is discussed. In section 3, the working of the proposed algorithm is discussed. The complexity of the algorithm is also discussed in this section. Next section i.e. 4, discusses the comparative analysis of the proposed algorithm with the existing ones. The last section gives a conclusions about the performance and hence, the results achieved by the proposed algorithm.

2. RELATED WORK

2.1 Quick Sort

Quick Sort is one of the most efficient internal sorting algorithm which is based upon the Divide and Conquer approach [7] and is the method of choice for many application because it uses less resources as compare to other sorting algorithm. To sort an array, it picks the pivot element and partitions the array into two parts, placing small elements on the left and large elements on the right, and then recursively sorts the two sub-arrays and place the pivot element at its correct position.

There are many different versions of quick Sort that pick pivot in different ways.

1. Always pick leftmost element as pivot.
2. Always pick rightmost element as pivot
3. Pick a random element as pivot.
4. Pick median element as pivot.

The key process in quick Sort is partition() . work of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and place all smaller elements than x before pivot element, and put all greater elements than x after x. which should be done in linear time[5,9,12,14].

2.1.1 Partition Algorithm

The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we exchange current element with arr[i]. Otherwise we ignore current element.

Pseudo code: Quick Sort()

```
/* This function choose rightmost element as pivot. */
/* array[] --> Array to be sorted, l  --> Starting index, h  -->
Ending index */

int partition (int array[], int l, int h)
{
   int x = array[h];    // taking last element as pivot
   int small = (l - 1);  // Index of smaller element

   for (int j = l; j <= h- 1; j++)
   {
      // If current element is smaller than or equal to pivot
element
      if (array[j] <= x)
      {
         small++;    // increment index of smaller element in
array
         swap(&array[small],  &array[j]);  //  Swap  current
element with small index
      }
   }
   swap(&array[small + 1], &array[h]);
   return (small + 1);
}

void quickSort(int array[], int l, int h)
{
   if (l < h)
   {
      int pivot = partition(array, l, h); /* Partitioning pivot
index */
      quickSort(array, l, pivot - 1);
      quickSort(array, pivot + 1, h);
   }
}
```

Analysis:

Time taken by QuickSort in general can be written as following.

T(n) = T(k) + T(n-k-1) + (n)

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.
The time taken by QuickSort depends upon the input array and partition strategy.

Following are three cases.

2.1.2 Worst Case

The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

T(n) = T(0) + T(n-1) + (n)

which is equivalent to

T(n) = T(n-1) + (n)

The solution of above recurrence is (n2).

Solution of above recurrence is also O(nLogn)

Although the worst case time complexity of QuickSort is O(n2) which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

2.1.3 Time complexity

The time complexity of quicksort is O(n log n) in best and average case but in worst case(in already sorted array) is up to $O(n^2)$.

So the performance falls on already sorted/almost sorted lists if the pivot is not randomized.

2.1.4 Space complexity

Space complexity of quicksort is O(log n), taking into account the stack space used for recursion.

2.2 Merge Sort

MergeSort is a external sort which is also use Divide and Conquer approach like Quick Sort algorithm. The main

difference between these two sort is Quick Sort is internal sort as Merge sort is external sorting. A Merge Sort Works as Follows:-

1. It divides input array in two halves using merge sort ,then calls itself for the two halves and then merges the two sorted halves.
2. The merge() function is used for merging two halves. The merge(array, l, m, r) is key process that assumes that array[l..m] and array[m+1..r] are sorted and merges the two sorted sub-arrays into one.

The following diagram from wikipedia shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged [13, 15].

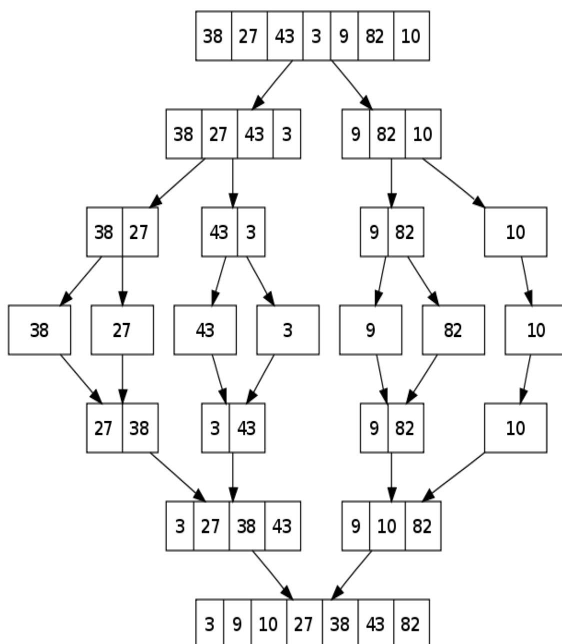

Fig 1: Diagram showing working of merge sort

2.2.1 Pseudo Code: Merge Sort()

```
 /* Function to merge the two haves array[l..m] and
array[m+1..r] of array array[] */
void merge(int array[], int l, int m, int r)

{
   int i, j, k;
   int n1 = m - l + 1; // no of elements in first temp subarray
   int n2 =  r - m;    // no of elements in second temp subarray

   /* create temp arrays */
   int L[n1], R[n2];

   /* Copy data to temp arrays L[] and R[] */
   for(i = 0; i < n1; i++)
      L[i] = array[l + i];
   for(j = 0; j < n2; j++)
      R[j] = array[m + 1+ j];

   /* Merge the temp arrays back into arr[l..r]*/
   i = 0;
   j = 0;
   k = l;
   while (i < n1 && j < n2)
   {
      if (L[i] <= R[j])
      {
         array[k] = L[i];
         i++;
      }
      else
      {
         array[k] = R[j];
         j++;
      }
      k++;
   }

   /* Copy the remaining elements of L[], if there are any */
   while (i < n1)
   {
      array[k] = L[i];
      i++;
      k++;
   }

   /* Copy the remaining elements of R[], if there are any */
   while (j < n2)
   {
      array[k] = R[j];
      j++;
      k++;
   }
}

/* l is for left index and r is right index of the sub-array
  of arr to be sorted */
void mergeSort(int array[], int l, int r)
{
   if (l < r)
   {
      int m = l+(r-l)/2;
      mergeSort(array, l, m);
      mergeSort(array, m+1, r);
      merge(array, l, m, r);
   }
}
```

2.2.2 ANALYSIS Time and Space Complexity

Given two sorted sub-arrays, together having a total of n elements, the merge operation uses an auxiliary array of size n to merge them together into a single sorted array in linear time i.e. O(n) in Big Oh notation.

Mergesort operation works as follows. Given an array of elements A, it sorts the left and right sub-arrays using mergesort itself, and then merges them together into one single sorted array. The base case is a sub-array of size 1, which is implicitly sorted.

If we analyze the time complexity of mergesort, we will see it is O(n log n) in all cases. That is, the time taken to sort n elements grows proportionally to n log n. Merge sort also needs an extra array of size n for the merge operation. So its space complexity is O(n).

2.2.3 Auxiliary Space: O(n)

2.2.4 Algorithmic Paradigm: Divide and Conquer

3. PROPOSED ALGORITHM: MQ SORT

We introduce a new algorithm MQ Sort to solve the problem by divide and conquer strategy, that is, in each partition one element pivot has located in correct position, which will be recursively executed until atmost 10 element left in array. Then merge sort will be called to sort the subarray of atmost 10 element. The theoretical evaluation shows Quick Sort takes time in order of O(n2) in worst case as Bubble Sort, Insertion Sort, Selection Sort, and takes time in order of O(nlogn) in best case as Merge Sort in both best and worst case. We can make MQ Sort pursuit best case by combining Quick Sort and Merge Sort. so that we can decrease the time complexity in order of O(nlogn). Furthermore the process of partition is finished by until we got subarray of atmost 10 element. Quick Sort demand for low memory compared with Merge Sort. Finally the empirical evaluation demonstrates that the CPU time of MQ Sort time decrease as compare to merge and quick sort. More specific, according to the significance testing when array size is greater and equal 2000 the difference is significant, whereas when array size is less and equal 2000 they are insignificantly different. The drawback of Quick Sort is also patently, when the pivot chosen for each partition get close to median, the time required decrease, whereas when the pivot gets closer to maximum or minimum, the inverse is true.

MQ Sort will perform the merge sort; if numbers of elements are less than 10 otherwise quick sort will work. As quick sort gives better results when number of elements are large and Merge sort gives better results when number of elements are less. That's why proposed algorithm performs better as compare to merge and quick sort.

3.1 Pseudo Code: MQ Sort()

```
/* l is for left index and r is right index of the sub-array
  of arr to be sorted */
void MQSort(int array[], int p, int r)
{
   if (p < r)
   {
            If(r-p<10)
      mergeSort(array,p,r);
            else
            {
            q->partition(array,p,r);
            MQsort(array,p,q-1);
            MQsort(array,q+1,r);

            }
   }
}
```

4. COMPARATIVE ANALYSIS

4.1 Time and Space Complexity

As mentioned above MergeSort and QuickSort both uses the divide and conquer approach. But Both algorithm have the some advantages and disadvantages.

Quick Sort is the In Place Sorting thus saving the performance, it doesn't require extra space but in worst case (Already Sorted array) due to wrongly selected pivot element, its complexity goes up to O(n2).

In Merge Sort, it require additional scratch space but it always gives O(n*logn) Performance(best, average and worst) in all the cases and it is a stable sort, and there is no worst-case scenario.

So In MQ sort we try to combine merge and quicksort to achieve the good performance as compare to merge and quick sort. Average case time: O(n*log(n)), However there are various differences.

So MQ sort always gives O(n*logn) complexity in all the cases like merge sort and do not use extra space like Quick sort, as we are using Quick sort for dividing the array if subarray is having more than 10 elements.

4.2 TIME COMPARISON OF MERGE, QUICK AND MQ SORT

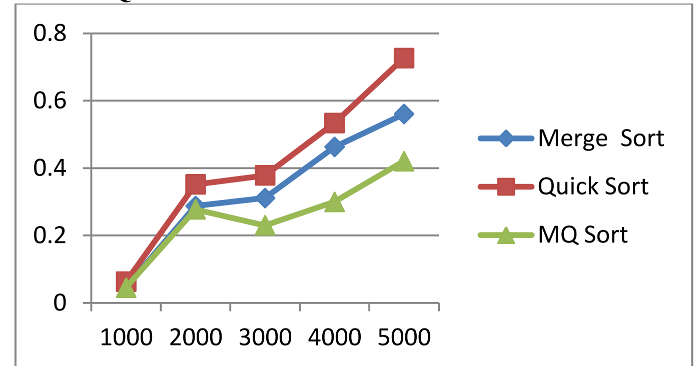

Fig 2: Chart showing time comparison of Quick sort, Merge sort and MQ sort

- The x-axis represents the number of elements sorted.
- The x-axis is linear and the last entries show the highlight feature of the graph. The difference in the three algorithms performance.
- MQ Sort seems to not want to let go off the axis. It is not a mistake, it is a reality. A sweet reality.

5. RESULTS AND DISCUSSIONS:

This Table Shows the time comparison of Merge Sort and Quick Sort with new introduced algorithm MQ Sort.

From Results we can see that MQ Sort take less time as Compare to Merge and Quick Sort, As it combine the Advantages of both the Algorithm. It is also based on Divide and Conquer approach.

Table 1. Table showing time taken to sort arrays of different sizes by Quick sort, Merge sort and MQ sort

Size of Array	Merge Sort	Quick Sort	MQ Sort
1000	0.0464	0.0630	0.0451
2000	0.2873	0.3518	0.2776
3000	0.3110	0.3784	0.2302
4000	0.4628	0.5336	0.2996
5000	0.5605	0.7263	0.4202

6. CONCLUSION

Table 2. Table comparing Quick sort, Merge sort and MQ sort

Name	Best Case	Average Case	Worst Case	Stable	Remark
Quick Sort	O(nlogn)	O(nlogn)	O(n2)	No	In Place Sorting
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	Yes	Require Extra Memory
MQ Sort	O (nlogn)	O(nlogn)	O(nlogn)	Yes	Does not Require Extra Memory

Any sorting algorithm for 10 integers will take negligible time and memory.

I would conclude this, that

- Merge Sort is suitable in External Sorting circumstance. It is an O(n*log(n)) algorithm in all cases , but it takes the extra memory.
- Quick Sort is an O(n*log(n)) algorithm on an average case and an $O(n^2)$ algorithm in the worst case scenario. (Quick sort's worst case occurs when the numbers are already sorted!!) The graph speaks it all. You need this algorithm when the list is large and time is premium. No Extra Memory is required.
- MQ Sort also is an O(n*log(n)) algorithm in all cases , but it does not takes the extra memory.

Quick Sort gives better results while list of elements is large and merge Sort gives better results when number of elements are less. Thus we proposed an algorithm named MQ Sort which is the combination of the two sorting algorithms i.e. Quick Sort and Merge Sort. Comparisons are also made with respect to time increasing the size of array.

7. REFERENCES

[1] Donald E. Knuth et al. "The Art of Computer Programming," Sorting and Searching Edition 2, Vol.3.

[2] Cormen et al. "Introduction to Algorithms," Edition 3, 31 Jul, 2009.

[3] D. Knuth, "The Art of Computer programming Sorting and Searching", 2nd edition, Addison-Wesley, vol. 3, **(1998)**.

[4] A. D. Mishra and D. Garg, "Selection of the best sorting algorithm", International Journal of Intelligent Information Processing, vol. 2, no. 2, **(2008)** July-December, pp. 363-368.

[5] C. A. R. Hoare, Algorithm 64: Quick sort. Comm. ACM, vol. 4, no. 7 **(1961)**, pp. 321.

[6] Ahmed M. Aliyu, Dr. P. B. Zirra, "A Comparative Analysis of Sorting Algorithms on Integer and Character Arrays," The International Journal Of Engineering And Science (IJES)., ISSN(e): 2319 – 1813 ISSN(p): 2319 – 1805.

[7] E. Horowitz, S. Sahni and S. Rajasekaran, Computer Algorithms, Galgotia Publications.

[8] Horowitz, E., Sahni. S, Fundamentals of Computer Algorithms, Computer Science Press, Rockville. Md

[9] Laila Khreisat, "Quick Sort: A Historical Perspective and Empirical Study", IJCSNS

[10] T. H. Coreman, C. E. Leierson, R. L. Rivest and C. Stein, Introduction to Algorithms, 2nd edition, MIT Press.

[11] John Darlington, Remarks on "A Synthesis of Several Sorting Algorithms", Springer Berlin / Heidelberg, pp 225-227,Volume 13, Number 3 / March, 1980.

[12] http://www.geeksforgeeks.org/iterative-quick-sort/

[13] https://en.wikipedia.org/?title=Merge_sort

[14] https://en.wikipedia.org/?title=Quicksort

[15] http://www.geeksforgeeks.org/forums/topic/merge-sort/