Quick Sort Optimized for Non-decreasing Data set

Omar Khan Durrani
Dept.t of Computer Science & Engineering,
Ghousia College of Engneering,
Ramanagram
omardurrani2003@yahoo.com

Sayed Abdulhayan
D*epartment of Computer Science and Engineering,*
PACE, Mangalore,
Karnataka, India
sabdulhayan.cs@pace.edu.in

***Abstract*-The Sorting Algorithm proposed by C.A.R Hoare in 1961 by name Quick sort, which is popularly known for being the fastest sorting algorithm. Quick sort is still being practiced in the field of computers systems and its applications. The Algorithm whose efficiency to sort random data set is represented in asymptotic notation as O(n log$_2$ n) and when quick sort algorithm has a input data set which is already Ordered then it takes a quadratic execution time which is considered as a worst case performance and this behavior is represented in asymptotic notation as O(n^2). The worst case performance is due the scan over heads which occur over the pre-sorted data set, in other words the partitioning gets skewed due to recursive calls and hence results in a quadratic complexity. This research paper presents an algorithm which minimizes a worst case execution time making it linear when the input list is in non-decreasing order. The paper describes how the improvements are accommodated in the existing quick sort. A priori analysis of proposed algorithm for different cases is made along with a proof of correctness. Later the algorithm is verified for its correctness and asymptotic performance. The algorithm is implemented using C++ and also we have compared with other popular quick-sort version.**

***Keywords*-Worst case, Presorted data, Linear time complexity, Early exit, part_no, Aorder status flag, Qwimb sort.**

I. INTRODUCTION

Problem Solving is one of the prime activity involved in the life cycle of human being and tools were the aids produced by him from the nature, starting from stone, sticks, fire,until it progressed to machinery and finally computers and artificial intelligence. As we know that a computer works on basis of Von Neumann's concept which is nothing but stored program which works to interface humans for problem solving. The generic name for a program is an Algorithm**.** An Algorithm is step-by-step procedure for solving a problem in a finite amount of time. Among the different types of algorithms, Sorting is frequently and widely used. In a computer, a sorting algorithm is that which puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. Mathematically, sorting is defined in [3] as as a list of numbers, A=(a_1,a_2,a_3.....a_n) which on permutation (reordering) results as, A=(a'_1,a'_2,a'_3.....a'_n) of the input sequence such that a'_1 <=a'_2, <=a'_3.....<=a'_n . The sequences are typically stored in arrays which are consecutive storage locations in the computer memory, the numbers are also referred to as the keys .

Since the dawn of computing, the Sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, Bubble sort was analyzed as early as 1956. later other algorithms like selection sort, insertion sort, shell sort, radix sort, and many others methods were introduced. Among these the popular once which are frequently used in system softwares and application software are merge sort, quick sort, insertion sort, which we term as popular sorting algorithms. Due to fast evolution in computer technology, although many consider sorting as a solved problem, useful new sorting algorithms are still being invented. Among the popular sorting algorithms Quicksort is the fastest one used in real world sorting problems. Quicksort uses a divide and conquer technique for sorting the given data file. We find sufficient work with respect to partitioning the unsorted data, selection of the Pivot element which divides the list and also regarding finding size of sub files. But fever work is found regarding improving the worst-case behaviors of Quick-sort algorithm which has been focused in this research study. In this paper we have presented a version of Quick-sort named as Qwimb sort which improves the worst case of the Hoare's Quick-sort [5] and the later Quick sort version found in[23].

II. QUICK_SORT ALGORITHM

Quick Sort is an algorithm based on the Divide and Conquer paradigm that selects a pivot element (in our example it is the left most element in the list) and reorders the given list in such a way that all elements smaller to it are on left side and those bigger than pivot is on the right side. Further the sub lists are recursively sorted until the list gets completely sorted as shown in Figure 1. Also you observe the Pivot element occupying its position as part of list being sorted. The time complexity of this algorithm is O (n log n) on un-ordered data set due to partitioning into two parts, along with two scans in opposite directions.

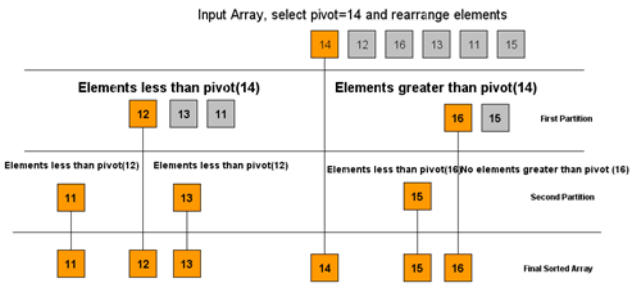

Fig. 1. shows **partitioning** using leftmost element as Pivot

A. Analysis of Quick-sort

The total time taken to re-arrange the array as described in the above section always takes O (n) or αn where α is some constant needed to execute in every partition. Let us suppose that the pivot we just choose has divided the array

into two parts: one of size k and the other of size n − k. Notice that both these parts still need to be sorted. This gives us the following relation:

$$T(n) = T(k) + T(n-k) + \alpha n \quad (1.1)$$

where T (n) refers to the time taken by the algorithm to sort n elements and α is the constant computation time for processing n elements to partition the list into k and n-k parts.

In order to analyze for the **worst case**, consider when pivot is the least element of the array (input array is in ascending order), so that we have k = 1 and n − k = n – 1 in (1.1). In such a case, we have:

$$T(n) = T(1) + T(n-1) + \alpha n$$

by solving the recurrence as follows:

$$= T(n-i) + iT(1) + \alpha . \sum_{j=0}^{i-1}(n-j) \quad (1.2)$$

Now clearly such a recurrence can only go on until i = n − 1 (because otherwise n – 1 would be less than 1). So, substitute i = n − 1 in (1.2), which gives us:

$$T(n) = T(1) + (n-1)T(1) + \alpha \sum_{j=0}^{n-2}(n-j)$$

on further simplification we arrive as shown below,

$$T(n) = nT(1) + \alpha (n(n-2) - (n-2)(n-1)/2)$$

which is O (n^2).This is the worst case of quick-sort, which happens when the pivot we picked turns out to be the least element of the array to be sorted, in every step (i.e. in every recursive call). A similar situation will also occur if the pivot happens to be the largest element of the array to be sorted.

The **best case** of quick sort occurs when the pivot we pick happens to divide the array into two exactly equal parts, in every step. Thus we have k = n/2 and n−k = n/2 in equation (1.1) for the original array of size n.

Consider, therefore, the recurrence:

$$T(n) = 2T(n/2) + \alpha n \quad (1.3)$$

$$= 2(2T(n/4) + \alpha n/2) + \alpha n$$

(Note: T (n/2) = 2T (n/4) + α n/2 by just substituting n/2 for n in (1.3)

$$= 2^2 T(n/4) + 2\alpha n$$ (By simplifying and grouping terms together).

$$= 2^2(2T(n/8) + \alpha n/4) + 2\alpha n$$

$$= 2^3 T(n/8) + 3\alpha n$$

$$= 2^k T(n/2^k) + k\alpha n$$ (Continuing likewise till the k^{th} step)

Notice that this recurrence will continue only until $n = 2^k$ (otherwise we have $n/2^k < 1$), i.e. until k = log n. Thus, by putting k = log n, we have the following equation:

$$T(n) = nT(1) + \alpha n \log n,$$

which is O (n log n). This is the best case for quick sort.

It also turns out that in the **average case** (over all possible pivot configurations), quick sort has a time complexity of O (n log n), the proof of which is commonly found in [1,3,4].

III. LITERATURE REVIEW ON QUICK SORT ALGORITHM

Thomas H. Cormen et. al. in [3] have quoted that Quick sort takes Quadratic time O (n^2) in the worst case, spends a lot of time on the sorted or almost sorted data. It performs about $n^2/2$ comparisons even on nearly sorted data found in[23], but swap count is low for sorted or almost sorted input. Mark Allan Weiss in [4] has also stated that quick sort has O(n^2) worst case performance. Howrowitz et al in [1] have said that, a possible input on which quick sort displays worst case behavior is one in which the elements are already in order.

Almost all the authors of Algorithm books and research papers showing quick sort analysis, have agreed that quick sort perform no better than O(n^2) in case of ordered input. With all the above survey and many others references the theoreticians and practitioners have considered Quick sort worst case to be classified in asymptotic class O (n^2). But it is found that there are some works which have encouraged to take up the study to improve in the worst case. A detail literature survey done with respect to Quick sort algorithm is presented in following paragraph before we present the design and implementation of our version Qwimb sort..

1961 C.A.R Hoare founded and implemented Quick sort **algorithm** as found in[5]. It was first written in Algol 60, which sorts the array in the random-access store of a computer, ie., a in-place sort. He also said that no extra space is required. The Pivot is randomly selected using a random number generator. In [6] Hoare has also indicated and described some places of refinement which can lead towards optimization. He analyzed his algorithm over random data and stated that the minimum number of comparisons required to achieve the reduction in entropy is $\log_2 N! \approx N\log_2 N$. The average number of comparisons required by quick sort is greater then the theoretical minimum by a factor $2\log_e 2 \approx 1.4$. He further suggested that the factor could be reduced by choosing Pivot as the median of a small random sample of the items in the segment. He further described the inner loop of partition could be optimized using a machine instruction to exchange elements. Mathematical analysis of algorithm was done using rules of inference. He also showed Merge sort running slower than Quick sort. The Function quicksort() of Algorithm 64 is recursive and its partition() function is iterative. Hoare did not speak about worst case and equal keys. He also claimed that no extra memory is needed. He showed a road-map for research in quick sort Algorithm. Later the claim made by Hoare was proved to be incorrect by B. Randell & J. Russell in [7].They showed that extra space was because of activation record needed by recursion. They also mentioned in their technical correspondence that no changes are required to quick sort and hence it works satisfactorily. Finally promoted Hoare's work.

Robert Sedgewick in his thesis titled "Quick sort" [14] which received outstanding thesis award under supervision of D. E Knuth has paid special attention to methods of mathematical analysis, which are used to demonstrate the

practical utility of the quick sort algorithm. He has developed an exact and efficient form of quick sort, further the same is derived for the average best case and worst case running times. Following improvements were made in Sedgewick's quick sort:

1. Pivot is Median of three elements.
2. To use insertion sort method for sort small subfile(for < 10 element).
3. Loop unwrapping is applied by using assembly language. This was analyzed and found optimal.
4. A analytical study of equal key is thoroughly done with implementation and testing.

Robert Sedgewick's research work has showcased a complete analysis especially from the point of priori estimates and the same was verified in his practical measurement as found in [16]. His experiments showed that quick sort is up-to twice fast then its competitors. It was told that small files to be avoided by being in recursion. Small file can use insertion sort. He also clearly says that, *quick sort needs $O(n^2)$ in worst case*. Practical measurement was less shown by the author. Robert Sedgewick made a unique way of study.

In November 1980 C.R Cook and D Jin Kim in [17] designed a Best Sorting Algorithm for Nearly Sorted Lists. The algorithm which is a hybrid version of quick sort, a combination of straight insertion sort, quicker sort and merge sort. The new algorithm performed well and showed improvement in sorting twice, when compared with same input with straight insertion sort ,shell sort, quick sort and heapsort. The author tested for sample size of 50, 800,200, 500, 1000, 2000 elements.

In April 1983, D. Motzkin in [18] presented a Algorithm which he called Meansort is based on quick sort. The Algorithm used a special technique at every partition in finding the Pivot. The algorithm name is based on the Mean value used as Pivot. The algorithm have improvements over the average cases. The algorithm was implemented in pascal language. Mean sort is considered improvement over standard quick sort. It is efficient even if repeating keys are present. Efficiency was measured based on interchanges comparison and particular stops. Means sort showed considerable improvement over quick sort.

In April 1984, an article found in [19] " How to sort " by Jon L Bentley, which showed in his experiments that the system sort, that is, qsort in UNIX Operating System was fast but performed a bit slow then the quick sort because it has to be called, qsort is version of quick sort which is made part of Unix system and hence used as a command to sort files. He also mentioned that System commands should fulfill the user needs and not all systems have the system sort command. The author did not mentioned about worst case of quick sort.

In 1985, Roger L. Wainwright in [20] showed a class of sorting algorithms based on quick sort. Bsort a variation of quick sort combines the interchange technique used in bubble sort with quick sort. The algorithm improved the average behavior of quick sort and claimed that the worst case situation of comparison for sorted or nearly sorted lists works best leading to removal of worst case ie., $O(n^2)$, which **later was proved incorrect by a technical correspondence** in [21]. it showed Bsort failed to be O(n) for a simple data set 2,4,6,8….n-2, 1,2,3….n-3 instead it took $O(n^2)$ time.

In 1987, Roger L. Wainwright in [22] modified Quick sort algorithms with an early exit for sorted subfiles. He also said that the improvements to quick sort have been made in the following areas:

1. Determining a better pivot value,
2. Considering the size of subfiles and
3. Schemes of partitioning the files.

Despite improvements the worst case still remains when a file is nearly or completely sorted, which can be assumed as a **4th area**. A version of quick sort called **qsorte** is presented that provides an early exit for sorted subfiles. He Tested on random files, sorted and nearly sorted and reverse sorted files. Results of quick sort, Quicker sort, Bsort, qsort are exhibited in experiments. qsorte perform good as quick sort for random files. Author has implemented using pascal language. Author says we should no longer refer to the quick sort algorithm as having a worst case behavior for sorted subfiles. He also showed that there are cases for which Quick sort has a worst complexity of $O(n^2)$.

In 1993, Jon L Bentley and M Douglas McIL Roy in [23] built a new qsort function for 'C' library based on Scowen's quicker sort choosing a partition elements by new scheme. Hoare's version for a case of **2n** integers 1 2 3 …n,n…..3 2 1 took n^2 comparisons to sort. The Authors have engineered the qsort version with needed improvements. The paper is fully supported by pioneers in the field of sorting. In this paper Program 7 used MACROS to improve its performances and used insertion sort for small sub arrays. Program 7 proved to be best. The authors have mentioned that, if worst case performance is important, Quicksort is the wrong algorithm.

In 2007, in the paper "Quicksort: A historical perspective and empirical study" [25] Laila khreisat studied and compared the Quick sort variants and the new algorithm of recent times. The study made was in terms of comparison performed and the running times on reverse order, already ordered and randomly generated orders. She tested on various random integers data set from N=3000 to 500000. The performances of all the variants are really interesting

Laila khreisat in 2018, presented Introsort found in [26] which is combination of qsorte and heapsort. When qsorte stops for stack overflow the sorting process is continued by heapsort. The paper has no information about worst case of $O(n^2)$ being eliminated.

In 2015, the research work in [28] showed the design and implementation of a modified version of quick sort algorithm named Quicksort_wmb, in which the algorithm had an early exit when ever it encountered the input array as presorted, This algorithm worked well for ordered data set (presorted as ascending or descending) and took linear time O(n) instead of quadratic $O(n^2)$. In recent research it is realized that the early exit leaves random arrays unsorted which had the first element as the least value. When the leftmost element in the input arrau is considered as pivot element.

A recent research of 2020 found in [29] by Bal, Aditi Basu and Chakraborty, Soubhik titled "An Experimental Study on a Modified Version of Quicksort" used the version quicksort_wmb published in [27] to experiment over various

continuous and discrete probability distributions and measured the performance in terms of the number of comparisons the algorithm makes to sort the whole array. A number of common probability distributions having both continuous and discrete-were simulated to constitute the elements of a random unsorted list of numbers. The modified version quicksort_wmb was applied on these arrays to sort the numbers. The results obtained were very interesting. The continuous distributions were sorted faster than the discrete ones by the algorithm, the reason for which, after investigation, was found to be the existence of ties in discrete distributions, thus providing an evidence that the version of quicksort is sensitive to ties. The sensitivity of quicksort_wmb to ties is not new. The interesting thing is that the sensitivity to ties remains irrespective of the improvement.

Finally as a summary of above detail survey it is observed that sufficient work has been done to improve speed for average cases especially in three areas, like determining pivot. considering the size of sub-files and schemes of partitioning, but as it is stated by Wainqright L.R in [22], a fourth area which is improvement to Worst case need to considered for research. It is also seen that $O(n^2)$ worst case of quick sort has not been improved much.

IV. Design of Qwimb Sorting

Mathematicians have contributed algorithmic analysis from info-theoretic view point, on the other side we, the algorithm engineers contribute from the angle of programming languages and the computer architecture. A responsibility that showers is to cope for the upcoming challenges to improve and provide a compatible code design keeping time efficiency as our objective. Quick sort is the only Algorithm which is a heart in this field. Also as mentioned by L. Wainwright in[22] where he suggested research regarding improvement in the worst case. Hence we considered Quick sort algorithm to analyze it and improve it especially in the **worst case.**

A. Avoiding Worst case

Practical implementations of quick sort often pick a pivot randomly from the list . This greatly reduces the chance of going into worst-case situation. This method of selecting pivot in random is seen to work excellently in practice but still much time is exploited by the randomizer [1] which matters the efficiency in execution. The other technique, which deterministically prevents the worst case from ever occurring, is to find the median of the array to be sorted each time, and use that as the pivot. The median can be found in linear time but that is saddled with a huge constant factor overhead, rendering it suboptimal for practical implementations [3].

In the present research we shall show a very simple version which performs good and improves in the performance when the input array is already in ascending order. This variant has also sorted the cases shown in[21]. This variant which we call as qwimb sorting, we make the early exit on two conditions, we check status of two pointers along with a check of orderliness of input array. The next section shows the design in detail.

A. Design Specification of Qwimb Sorting

As we know that quick-sort algorithm is based on the Divide-and-Conquer paradigm that selects a pivot element and reorders the given list in such a way that all elements smaller or equal to the Pivot are on one side and those bigger than it are on the other. Thus the sub lists are recursively sorted until the whole list gets completely sorted, also explained in section II when ordered input is considered, we need to perform best case, that is, in linear time. We have considered the first element of the array or list as the Pivot element.

To make this to happen we consider the leftmost element as the pivot. We have considered three **global integer variables, no_part and Aorder** which are initialized to zero, further the decision to early exit from the quick sort is made when the respective values of global variables ie.,no_part and Aorder equals to 1 and recursion is avoided later on. The global variable *no_part* counts the number of partitions made in each execution of quick sort, *Aorder* (set to 1) is an indicator when the array is encountered to be in ascending order.

Aorder is set to value 1 in the partition function when *i=1* and *j=0,* that is i does not get incremented further as key>=a[i] becomes false in the statement do{i++;}while(key>=a[i]); (do- while is executed only once), where as key<a[j] is true until *j* become 0 after *n* executions of the statement do{j--;}while(key<a[j]); (until key=a[j]). As shown in Figure 2 in case of ascending order input we can make a early exit.

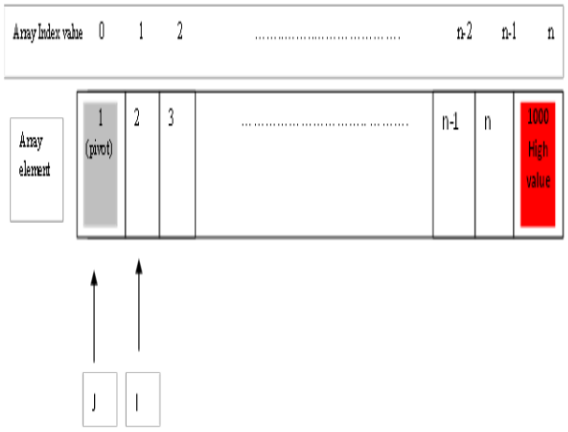

Fig. 2. shows the Position of index i & j at the end of first partition when the input array is in ascending order

Further the above changes when applied in Quicksort it will fail to sort the random input array which has the first element as the lowest among the array elements. In other words when the Pivot (leftmost element) is the least amongst all elements of input array. To avoid such cases, we add a small routine which checks for pre-sortedness and avoids an early exit because the Aorder flag shall not be set to 1, instead it continues to sort the random data set. This modification will take extra time of about O(n). The extra time is taken in case the array is presorted or if the element is the smallest element for the other case, otherwise it runs normally as the Hoare's version does it.

V. Implementation Qwimb Sorting

With respect to the study made in[28,30], we have selected C++ programming language to implement Qwimb sorting as it best suits being a general purpose programming language. C++ has exhibited the behaviors of sorting algorithms as per the priori estimates and priori analysis

found in[1,30]. The following sections shows the different segments of the C++ code implementation.

A. Global Declarations

The Global declarations as shown in Table 1 includes the preprocessor statements which are basically needed for input and output operations. The global static identifier no_part is used to count the number of partitions made during the sorting. Aorder is a global and is used as a status flag which is set when we find that the array is Presorted in Ascending order.

B. Aordered Function

The Aordered function in Table 2 scans from left most element of array and returns a boolean value **True,** if the array of input numbers are already in ascending order else a False is returned when it finds a element which is greater than the consecutive next number in the array.

C. Qwimb Recursive Function

The Qwimb function as shown in Table 3 recursively sorts the input array by calling partition_iwmb function given in Table 4 to partition the array into two with respect to the pivot element. The Qwimb function exits from the recursion when it finds the array to be in Ascending order with the help of partition count and Aorder flag status.

D. Partition_iwmb Function

The partition_iwmb function as shown in Table 4 assigns the first element as the pivot and divides the array into two parts. During the first partition it checks if the array is already sorted with the help of the i and j index values and the boolean value returned by Aordered function and sets the Aorder flag.

TABLE I. GLOBAL DECLARATION FOR C++ CODE

Line No.	Statements
1.	#include <cstdlib>
2.	using namespace std;
3.	static int no_part=0; //global Variable
4.	int Aorder=0; //global Variables

TABLE II. C++ CODE OF AORDERED FUNCTION

Line No.	Statements
1.	bool Aordered(int a[],int n)
2.	{for(int i=0;i<n;++i)
3.	if (a[i]>a[i+1])return false;
4.	return true; } // end of Aordered

TABLE III. C++ CODE OF QWIMB RECURSIVE FUNCTION

Line No.	Statements
1.	void Qwimb(int a[],int low,int high)
2.	{ int j;
3.	if(low<high)
4.	{ j=partition_iwmb(a,low,high);
5.	if (no_part= =1)
6.	if(Aorder){cout<<"Aorder"<<endl;return;}
7.	Qwimb(a.low,j-1);
8.	Qwimb(a,j+1,high);
9.	} // end of if compound statement
10.	} // end of Qwimb

TABLE IV. C++ CODE OF PARTITION_IWMB ITERATIVE FUNCTION

Line No.	Statements
1.	int partition_iwmb(int low,int high)
2.	{int key,i,j,temp;
3.	no_part++; // Global variable counts partitions
4.	key=a[low]; // assigning the pivot
5.	i=low; j=high; // initializing left & right pointer
6.	while(i<j)
7.	{while(a[i] <= key) i++;// hunts element > key
8.	while(a[j] > key) j--; // hunts element <= key
9.	if(i<j){temp=a[i];a[i]=a[j]; a[j]=temp;}
10.	} // end of while
11.	temp=a[low];a[low]=a[j];a[j]=temp; // swap j th element with pivot
12.	if ((i==1) &&(j==0)&& Aordered(high,n)) Aorder =1; // set flag if Ascending
13.	return j; //returns the pivot }//end of partition_seek

VI. POSTERIORI TESTING OF QWIMB SORTING

The Qwimb code is tested for its correctness on different input samples of data under following 3 categories:

1) Pseudo Random generated input numbers (Table 5)

2) Ordered input numbers (Table 6)

3) Special cases of input numbers (Table 7)

A. Program testing for random numbers and Non-decreasing numbers.

The procedure to generate Pseudo Random Numbers(PRNs) and Non-decreasing array of numbers is described in [28], further set ups like set number ,size number and range were initialized as per necessity in the driver segment of C++ program. Also number of partitions made by the algorithm is also highlighted in the output. Partition counts gives a complexity indication, for example if number of partitions less than the size of input array to 1 which indicates a average case complexity. Similarly when Number of partitions is 1, it indicates a best case complexity.

TABLE V. RESULT OBTAINED WITH PRNS

Set No	Range	Input Array elements(size= 5)	Sorted output elements	No. of Partitions
1	0-100	35 86 92 49 21	21 35 49 86 92	3
2	0-100	62 27 90 59 63	27 59 62 63 90	3
3	0-100	26 40 26 72 36	26 26 36 40 72	2
4	0-100	11 68 67 29 82	11 29 67 68 82	3
5	0-100	30 62 23 67 35	23 30 35 62 67	2

(Random data input shows a O(nlogn) complexity)

TABLE VI. TABLE 6. RESULT OBTAINED WITH NON-DECREASING NUMBERS

Set No	Range	Input Array elements(size= 5)	Sorted output elements	No. of Partitions
1	0 to13	:0 3 6 9 12	:0 3 6 9 12	1
2	-4 to 0	-4 -3 -2 -1 0	-4 -3 -2 -1 0	1
3	-1to3	-1 0 1 2 3	-1 0 1 2 3	1
4	-8 to 0	-8 -6 -4 -2 0	-8 -6 -4 -2 0	1
5	-12 to0	-12 -9 -6 -3 0	-12 -9 -6 -3 0	1

(Ascending ordered input shows a O(n) complexity due to exit after first partition.)

B. Program Testing on Special Input Cases

The special cases are those which are considered in [21] and has shown quadratic complexity, which Qwimb has done in nlogn time complexity. The Table 7 shows the results when special cases were considered. The results obtained for the above three categories of data, shows the correctness of the algorithm

TABLE VII. RESULT OBTAINED FOR SPECIAL CASES INPUT

Set No.	Input Array elements	Sorted Array elements	No. of Partitions
1.	4 9 5 6 10 2 8 3 7 1	1 2 3 4 5 6 7 8 9 10	6
2.	1 2 3 4 5 4 3 2 1 0	0 1 1 2 2 3 3 4 4 5	6
3.	1 2 3 4 5 1 2 3 4 5	1 1 2 2 3 3 4 4 5 5	6
4.	1 2 3 4 5 0 501 0 502 0	0 0 0 1 2 3 4 5 501 502	6
5.	2 4 6 8 10 9 7 5 3 1	1 2 3 4 5 6 7 8 9 10	6
6.	2 8 4 12 10 1 3 5 7 9	1 2 3 4 5 7 8 9 10 12	5

VII. ASYMPTOTIC PERFORMANCE MEASUREMENT OF QWIMB SORTING

A. Experiment Set-up and Data Generation

The Test bed for conducting our experiments has Memory of 3.7 GiB, Intel® CoreTM i3-6006U CPU @ 2.00GHz × 4 ,64-bit, 970.9 GB, Ubuntu 16.04 LTS. Coding Language used: linux system based compiler C++. The experiment Setup and data generation methods used are as specified in[28].

A. Performance Measurement

Table 8 and Table 9 shows the results of executions on random and ordered(ascending order) samples respectively in our system which is achieved by some modification made in the code. The output shows the data sample size, time of execution for sorting in seconds and the number of partitions made by Qwimb() sorting algorithm. For simplicity we have considered output range from 100,000 samples to 10,00,000 in steps of 100,000. It is observed from the output that we have fever partitions then the data sample size for random data set, which indicates the time complexity of O (nlogn). Similarly when executed for non-decreasing data set we have only one partition which indicates the time complexity of Θ (n).

TABLE VIII. EXECUTION TIME FOR PRNS DATA SAMPLES

Sample size	Time (s)	No_of_partitions
100000	0.029839	67751
200000	0.050392	135390
300000	0.076136	203115
400000	0.099895	271008
500000	0.125967	338624
600000	0.155165	406315
700000	0.182476	474123
800000	0.211204	541884
900000	0.237766	609667
1000000	0.266798	677134

TABLE IX. TABLE 10: EXECUTION TIME TAKEN FOR ORDERED DATA SAMPLES

Sample size	Time (s)	No_of_partitions
100000	0.002092	1
200000	0.001457	1
300000	0.002185	1
400000	0.002922	1
500000	0.003808	1
600000	0.004548	1
700000	0.005159	1
800000	0.005913	1
900000	0.00707	1
1000000	0.007428	1

Further, the experiments are conducted for different sorting algorithms like bubble sort, insertion sort, mergesort, Hoare's quicksort and Bentley's version ,along with our Qwimb sort for a comparative study. The sorting results on random data sets and ordered data sets are shown in tables 11 and 12. In-order to attain the Asymptotic behavior range we have considered the range of sample data set from 10000 to 500,000 in steps of 10000 till 100000 and in steps of 100000 onwards.

B. Observations

Considering the results shown in tables 11 & 12 listed below are the observations made:

1) Asymptotic Behaviors of popular sorting Algorithms are clearly experimented and explained in the research published in [28] therefore we concentrate more on the performance of our version of quick sort (qwimb sorting).

2) In both tables 11 & 12 Bubble sort takes the maximum time to sort as it belongs to quadratic class of growth function.

3) In table 11 for random data sets, Merge sort is slow when compare to Hoare's Quick sort. It is also seen that qwimb sort is a bit slower (negligible quantity)than Hoare's Quick sort. And as the optimal version QuickSort-3w by Robert Sedgewick and Bentley, found in [23] has performed faster. All of them belong to nlogn class of growth function.

4) In table 12 for ordered data sets, we see that execution time of both Hoare's quick sort and Bubble sort are nearly the same, witnessing that both have a quadratic time complexity i.e $O(n^2)$ as per their behaviors exhibited and priori analysis found in[1,28].

5) We see that when considering the size n=500,000 the execution time for Hoare's quicksort is 805.665 seconds, QuickSort-3w is 389.347 seconds and that of Qwimb is 0.1344417 seconds witnessing the complexities of Hoare's as quadratic, QuickSort-3w is far better than Hoare's where as Qwimb performing in linear time.

6) Finally, we see execution time of ordered data samples for Qwimb sorting is linear and that with with respect to Hoare's Quick sort and Bentley and Sedgewick's taking Quadratic complexity.

VIII. CONCLUSION

As per the literature review it is really found that sufficient work is done in improving the speed in the average case of quick sort and fever is found to improve the speed of execution for the worst cases, which occurs when the input list is presorted. Though some work is done but still has some special cases for which it lacks to speed up. The Qwimb sorting which we have proposed handles the ascending data sets taken as input and sorts in linear time as it takes only one partition. Qwimb sorting has also performed a bit faster even in the average cases when we have randomly generated numbers as input data set. The work also shows the mathematical analysis of Qwimb sorting. Also, program testing and experiments conducted verifies the priori analysis made for program. Performance measurement considering large data set is done so that we observe asymptotic behaviors of algorithm. Presently the Qwimb algorithm considers for ascending ordered and pseudo-random generated data sets as input, in future the work can also give consideration for decreasing ordered data sets as input hence making it a sorting algorithm of time complexity O(nlogn).

TABLE X. SHOWS TIME TAKEN FOR RANDOM DATA SAMPLES

Input Size	Quick Sort time(s)	Merge sort time(s)	Bubble sort time(s)	Quick Sort-3w time(s)	Qwimb sort time(s)
10000	0.010	0.003	0.403	0.00187	0.002352
20000	0.019	0.006	1.779	0.004057	0.004272
30000	0.013	0.009	4.102	0.006324	0.006825
40000	0.012	0.012	7.438	0.008894	0.009194
50000	0.013	0.016	11.844	0.011197	0.011541
60000	0.015	0.018	17.129	0.013297	0.013772
70000	0.020	0.022	23.586	0.015919	0.016738
80000	0.021	0.025	31.208	0.018413	0.019033
90000	0.024	0.029	39.373	0.020875	0.021631
100000	0.027	0.032	48.755	0.023114	0.024318
200000	0.0557	0.0697	195.58	0.048613	0.050938
300000	0.0864	0.1024	442.68	0.074816	0.078249
400000	0.1148	0.1385	786.60	0.100895	0.106358
500000	0.1493	0.1764	1226.79	0.129536	0.134417

TABLE XI. SHOWS TIME TAKEN FOR NON-DECREASING DATA SAMPLES

Input Size	Quick Sort time(s)	Merge sort time(s)	Bubble sort time(s)	Quick Sort-3w time(s)	Qwimb sort time(s)
10000	0.32702	0.00194	0.34876	0.08701	7.6e-05
20000	1.30349	0.00327	1.31706	0.13098	0.0001
30000	2.92996	0.00527	2.98455	0.23124	0.0002
40000	5.20315	0.00675	5.29415	0.29780	0.0002
50000	8.13089	0.01015	8.52272	3.89705	0.0003
60000	11.7192	0.01036	11.9832	5.61249	0.0004
70000	15.9394	0.01218	16.2143	7.6635	0.0005
80000	20.8201	0.01515	23.6834	9.9546	0.0005
90000	26.4484	0.01590	28.5272	12.655	0.0006
100000	32.6818	0.01901	33.6542	15.620	0.0007
200000	134.072	0.03866	133.496	62.100	0.0014
300000	295.033	0.05822	299.489	139.79	0.0022
400000	529.134	0.07896	530.479	249.14	0.0038
500000	809.665	0.09970	827.23	389.34	0.0043

REFERENCES

[1] Sartaj Sahni, Data structures and Algorithms in C++, university press publication 2004, chapter 4 performance measurement, pages 123-136. https://books.google.co.in/books? id=QeVtQgAACAAJ}

[2] Levitin, A. Introduction to the Design and Analysis of Algorithms. Addison-Wesley, Boston MA, 2007. https://books.google.co.in/books? id=pOZSS9YeJYkC

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest,Clifford Stein, Introduction to Algorithms, Second Edition, Prentice-Hall New Delhi,2004. https://books.google.co.in/books ?id=YsE2pwAACAAJ},

[4] Weiss, M. A., Data Structures and Algorithm Analysis in C,Addison-Wesley,Second Edition, 1997 ISBN: 0-201-49840-5.https://books.google.co.in/books? id=dG8YZ521YVYC

[5] C.A.R. Hoare. "Algorithm 64: Quicksort." Communications of the ACM, vol. 4, pp. 321, Jul.1961. doi/10.1145/366622.366644

[6] C.A.R. Hoare. "Quicksort." Computer Journal, vol. 5, 1962, pp. 10–15. https://academic.oup.com/comjnl/article/5/1/10/395338

[7] B.Randell & J. Russell. "Certification of algorithlns 63, 64, 65: Partition, Quicksort, Find . " Comm. ACM ,July 1961. https://dl.acm.org/doi/10.1145/366707.367561

[8] R.S. Scowen, "Algorithm 271: Quickersort," Comm. ACM 8,11, pp 669-670, Nov. 1965. doi:10.1145/365660.365678

[9] BLAIR, C. R. "Certification of Algorithm 271: Quickersort". Comm.ACM,9,5(May1966),354. https://doi.org/10.1145/355592.365613

[10] Van Emden, M.H. "Increasing the Efficiency of Quick-sort". Communications of the ACM 13, 9 (September 1970), 563-567. https://doi.org/10.1145/362736.362753

[11] Van Emden, M.H. "Algorithm 402, qsort ",Comm. A C M 13, 11(Nov. 1970), 693-694. https://doi.org/10.1145/362790.362803

[12] M. Foley and C.A.R Hoare, "Proof of a rccursive program: Quicksort". Computer . Journal. 14, 4 (Nov. 1971), 391-395.https://doi.org/10.1093/comjnl/14.4.391

[13] R. Loeser. "Some performance tests of: quicksort: and descendants." ,Communications of the ACM, vol. 17, Mar. 1974, pp. 143–152.

[14] R. Sedgewick. "Quicksort." PhD dissertation, Stanford University, Stanford, CA, USA, 1975.

[15] Sedgewick, R. "Quicksort With Equal Keys". SIAM Journal of Computer, 6,2 (June 1977) 240-267. https://sedgewick.io/wp-content/themes/sedgewick/papers/1977Equal.pdf

[16] R. Sedgewick. "Implementing Quicksort programs", Communications of the ACM, vol. 21(10), pp. 847–857, 1978. https://doi.org/10.1145/359619.359631

[17] Cook, CR.. and Kim, D.J. "Best sorting algorithm for nearly sorted lists". Communications of the ACM, 23, 11 (Nov. 1980), 620-624. https://doi.org/10.1145/359024.359026

[18] Motzkin. D. "Meansort", Communications of the ACM, 26,4 (Apr. 1983), 250-251. https://doi.org/10.1145/2163.358088

[19] J. Bentley. "Programming Pearl: How to sort.", Communications of the ACM, vol. 27(4),1984. https://doi.org/10.1145/3894.315111

[20] R.L. Wainwright. "A class of sorting algorithms based on Quicksort", Communications of the ACM ,vol. 28(4), pp. 396-402, April 1985. https://doi.org/10.1145/3341.3348

[21] CORPORATE "Tech Correspondence, Technical correspondence." ,Communications of the ACM, vol. 29(4), pp. 331-335, Apr. 1986, https://doi.org/10.1145/5684.315618

[22] R.L. Wainright. "Quicksort algorithms with an early exit for sorted subfiles." ,Communications of the ACM, pp. 183-190, 1987. https://doi.org/10.114

[23] J. L. Bentley, M. D. McilRoy, "Engineering a Sort Function" ,Software—Practice and Experience, Vol. 23(11), Nov. 1993, pp 249 – 1265 . https://doi.org/10.1002/spe.4380231105

[24] J.L. Bentley, R. Sedgewick, R. "Fast algorithms for sorting and searching strings" ,in Proc.8th Annual ACM-SIAM symposium on Discrete algorithms, 1997, pp. 360–369.https://dl.acm.org/doi/10.5555/314161.314321

[25] L. Khreisat. "QuickSort: A Historical Perspective and Empirical Study", International Journal of Computer Science and Network Security, vol. 7(12), Dec. 2007. http://paper.ijcsns.org/07_book/200712/20071207.pdf

[26] L. Khreisat. "A Survey of Adaptive QuickSort Algorithms",international Journal of Computer Science and Security (IJCSS), Volume (12) : Issue (1) : 2018. https://www.cscjournals.org/library/manuscriptinfo.php?mc=IJCSS-1372

[27] O.K. Durrani, S.A.K. Nazim, "Modified quick sort: worst case made best case". International Journal for Emerging Technology and Advances Eng. 5(8). Website: www.ijetae.com . ISSN 2250-2459, August 2015.

[28] O. K. Durrani, A. S. Farooqi, A. G. Chinmai and K. S. Prasad, "Performances of Sorting Algorithms in Popular Programming Languages," 2022 International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON), Bangalore,india,2022,pp.1-7. doi:10.1109/SMARTGENCON56628.2022.10084261

[29] Bal, A.B., Chakraborty, S. (2020). "An Experimental Study of a Modified Version of Quicksort". Advances in Computational Intelligence. Advances in Intelligent Systems and Computing, vol 988.Springer,Singapore. https://link.springer.com/chapter/10.1007/978-981-13-8222-2_27

[30] Omar khan Durrani and Dr. Sayed Abdulhayan , "Asymptotic Performances of Popular Programming Languages for Popular Sorting Algorithms", Semiconductor Optoelectronics,Vol. 42 No. 1 (2023), 149-169, https://bdtgd.cn/article/view/2023/149.pdf

First Author Name: Dr. Omar Khan Durrani
Assist. Prof. & Head of Dept. ISE, Ghousia College of Engineering ,Ramanagram.
Education: Bachelor's Degree in CSE from National Institute of Engineering, University of Mysore , Mysore.
Master's Degree in CSE from National Institute of Technology Karnataka, Surathkal, Deemed University., Ph.D from VTU, Belgaum, Karnataka.
Teaching Experience: 28 years.
Research Areas : Algorithm Design and Analysis in Searching and Sorting, Internet of things.
Publications: 13 papers(journals & Conferences. 01 **Patent** filled. **Citations**: 62
Awards: Three times Best paper awards, Classic reviewer for IEEE Conference(CMT) and Asian Jounal, Awarded High Impact teaching Skills by Dale Carnegie and Wipro's Mission10x.

Second Author Name : Dr. Sayed Abdulhayan

Professor, Dept. CS&E, PACE Mangalore. Head for Academic Industry Relationship, Incubation center, Research activities and Entrepreneurship Development.
Education: B.E(ECE), M.Tech(Digital Communication and Networking), PhD(QoS and Security in 4G LTE-Advanced).
Total Teaching Experience: 18 years including 3 years of industrial experience.
Research Areas : Wireless communication, Mobile communication, Antenna synthesis, Cloud Computing, 5G/6G, algorithms,DBMS.**PhDs Guided: 0**4 Candidates.
Published papers : 44 papers in international journals and Conferences.Patents 01 approved & 05 filled.**Citation :38**
.**Professional memberships**: Member of Computer Society India(CSI), Member of Institute for Engineering Research and Publication (IFERP).

First Author

Second Author