Optimizing Complexity of Quick Sort

Md. Sabir Hossain[1], Snaholata Mondal[1], Rahma Simin Ali[1], and Mohammad Hasan[2](✉)

[1] Chittagong University of Engineering and Technology, Chattogram, Bangladesh
[2] Bangladesh Army University of Science and Technology, Saidpur, Bangladesh
hasancse.cuet13@gmail.com

Abstract. Quick Sort is considered as the fastest sorting algorithm among all the sorting algorithms. The idea of selecting a pivot was introduced in classical Quick Sort in 1962. This sorting algorithm takes favorably less time compared to other methods. It needs a complex time O(*nlogn*) for the best case and O(n^2) for worst-case which occurs when the input array is already sorted or reversely sorted. To reduce the worst-case complexity we provide a strong algorithm where it makes fewer comparisons and the time complexity after using this algorithm becomes a function of logarithm O(*nlogn*) for worst-case complexity. We experimentally evaluate our algorithms and compare them with classical algorithms and with other papers. The algorithm presented here has profound implications for future studies of handling worst-case complexity and may one day help to solve this occurrence of the fastest sorting method.

Keywords: Quick Sort · Reversely sorted · Time complexity · Best case · Worst case · Logarithm

1 Introduction

Quick Sort is one of the most efficient sorting algorithms. It is capable of sorting a list of data elements comparatively faster than any of the common sorting algorithms. Quick sort is also called as partition exchange sort. It is based on the splitting of an array into smaller ones. Basically it works on the idea of divide and conquer rule. It divides the array according to the partition function and the partition process depends on the selection of pivot. Its worst case complexity made this fastest algorithm a little bit vulnerable. Many authors researched for reducing the worst case complexity O(n^2) either to O(n) or to O(*nlogn*). In [1] author optimized the complexity of Quick Sort algorithm to O(n) using Dynamic Pivot selection method. The author R. Devi and V. Khemchandani in paper [2] have used Median Selection Pivot and Bidirectional Partitioning to reduce the worst-case complexity. The general Quick Sort algorithm takes O(n^2) time. Here, in this paper, we presented an algorithm which divides the array as half portion to calculate the pivot and then we use this pivot in partition function that divides the main input array. After recursive calling of the quicksort again and again finally we get the sorted output of input array. Although this algorithm is not unique as we took help from various papers and sources. The paper includes some section describing the

© Springer Nature Singapore Pte Ltd. 2020
N. Chaubey et al. (Eds.): COMS2 2020, CCIS 1235, pp. 329–339, 2020.
https://doi.org/10.1007/978-981-15-6648-6_26

whole project more significantly. Section 2 provides the discussion of the related work with associated limitations. Section 3.1 describes the outcomes of this research paper. Section 3.2 includes the preliminaries where the main idea of Quick Sort has been described. Section 3.3 describes the overview of proposed methodology including a diagram explaining the whole process. Section 3.4 shows the algorithm of proposed methodology in details including the pivot selection. Section 5 discusses the experimental results of the proposed method. Finally, Sect. 6 concludes the paper and shows the future directions to this paper.

2 Related Works

Jaja, A. et al. [3] has mentioned the partitioning process of the QuickSort algorithm. The process of a randomized selection of pivot has been discussed in detail. But the limitation is that the proof of randomized Quick Sort is difficult to understand. The basics of Quick Sort where the storage of computers has been given the priority and basic algorithm has been discussed on paper [2]. The key contributions are partition, comparison of quicksort with merge sort and cyclic exchange. But the limitation is that it says nothing about reducing the worst-case complexity of Quick Sort.

Paper [4] also discusses the preprocessing of large data sets using the QuickSort algorithm. Here, the method of random reference selection has been used. Key contributions are the comparison of complexity, handling of the large input set. But the random reference element selection method is difficult to understand and implementation of this method has not yet been discussed and these are counted as big limitations.

Paper [5] has reduced the complexity of the worst case of Quicksort algorithm to O(n) from O($nlogn$) for unsorted arrays and Dynamic Pivoting method has been used. The key contribution is the median of five/seven/nine. Random pivot selection, recursive calls, Boolean variable to see if the array is already sorted. The limitation is that as per empirical analysis the proposed algorithm could not run with O($nlogn$) complexity. Paper [6] depicts an overview of the pivot selection method. A new pivot selection method Median of five has been introduced. Paper [1] discusses improving the complexity of quick sort algorithm where key contributions are dividing the array into two equal halves and dynamic pivoting. Dynamic pivoting, Recursive calls, Boolean variables are the basic contributions. But the paper could not prove the experimental research.

Many parallel sorting algorithms among which three types of algorithm and their comparative analysis has been discussed by author Sha, L. in Paper [7] and Singh Rajput, I. et al. in paper [11]. The analysis has taken place based on average sorting time, parallel sorting and comparing the number of comparisons. Two versions of Quick Sort the classical and the proposed one has been discussed by Devi, R. et al. In[2]. The worst-case running time of quicksort has been discussed and reduced to O($nlogn$) from O(n^2). In paper [8] pivot based selection technique has been discussed and the dynamic selection of pivot has been introduced. In this paper [8], a new dynamic pivot selection procedure has been introduced that allows the database to be initially empty and grow later. Lakshmi, I. et al. in paper [9] discusses four different types of basic sorting algorithm where sorting algorithms are compared according to In paper [6] tried to explain controlling complexity is the simplest way possible and to do this a simple reliability model has been introduced.

Conceptual framework, forward recovery solution, N version programming are the key contributions. But the explanation of controlling complexity is hard to implement and neither assumption is easy in practice.

Schneider, K. et al. in paper [7] described a Multiway Quicksort to make the algorithm more efficient which gives reason to believe that further improvements are possible with multiway Quicksort. Aumüller, M. et al. in paper [8] explained multi-pivot Quick Sort which refers to a variant of a classical quicksort wherein the partitioning step k pivots are used to split the input into k + 1 segments.

Aumüller, M. et al. in paper [9] tried to introduce a model that captures all dual-pivot algorithms, give a unified analysis, and identify new dual-pivot algorithms that minimize the average number of key comparisons among all possible algorithms and explained that dual-pivot quicksort benefits from a skewed choice of pivots. In paper [10], Kushagra, S. et al. proposed a 3-pivot variant that performs very well in theory and practice that makes fewer comparisons and has better cache behavior than the dual-pivot quicksort.

Authors Faujdar N and Ghrera P in paper [11] showed an evaluation of quick sort along with merge sort using GPA computing which includes the parallel time complexity and total space complexity taken by merge and quick sort on a dataset. In paper [12], authors showed an comparison of parallel quick sort with the theoretical one. A special kind of sorting which is double hashing sort can be known with the help of paper [13]. With the help of paper [14], optimized selection sort and the analysis of the optimization process can be understood very well. A dynamic pivot selction method is presented in a very well method on the paper [15].

3 Methodology

The authors of many papers tried in many ways to reduce the complexity of Quick Sort using different ways. In this algorithm worst case of Quick Sort has been modified by calculating mean as the pivot element. The pivot selection method has been done by dividing the array into two sub-array of equal size. Then the maximum and minimum element of both sub-array is calculated. The average value of all these four values is considered as pivot element. Then the partitioning is happening by comparing each element of both sub-array with the pivot element. Thus the loop will be running half of the array only. Here, if the element of the right subarray is smaller than the pivot, the loop variable will increment. Similarly, if the element of the left subarray is greater than the pivot, the loop variable will decrement. After that swap function is called. When the size is equal or less than 3 then the Manual Sort procedure is called which is actually a compare between the elements. As there is no loop in this function, the time is reduced because the recursion function is not called. Thus this algorithm does not lead to the worst case of $O(n^2)$ and it becomes near to $O(n)$ (Fig. 1).

Here in the above flow chart, the algorithm has been presented in short. When the QuickSort function is called, it checks whether the input size is greater than 3 or not. If input size is greater than 3 then it calculates pivot taking the average of maximum and minimum values from both sub-array. After calculating the pivot, the partition function is called where the values are compared with pivot. After completing all the functions, we get a sorted array which is our desired output.

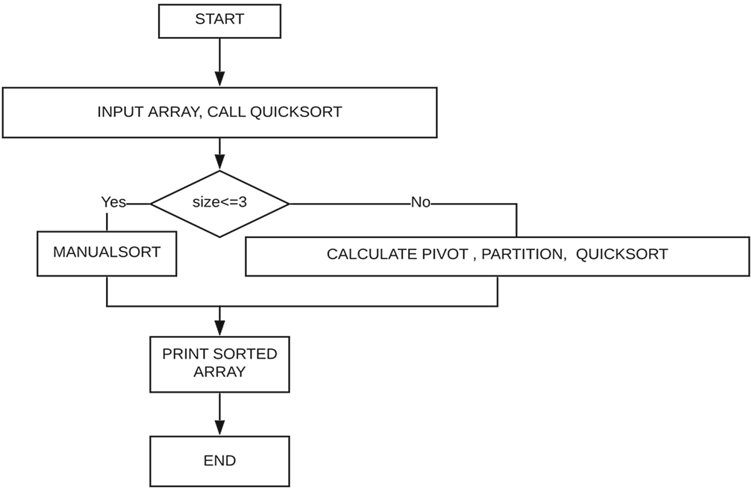

Fig. 1. Flow chart of our proposed algorithm.

3.1 Pseudocode of the Classical Algorithm

The classical algorithm consists of two portions. The main function Quick Sort is called in the first portion where the last element is selected as pivot and passed as an argument in the second portion which is partition function where each element is compared with the pivot i.e. last element.

Algorithm 1: CLASSICAL QUICK SORT.

```
Classical_Q_Sorter (ARRA,PI,RI)
  1.   if (PI < RI)  then
  2.     qt ← Partition(ARRA,PI,RI)
  3.     Classical_Q_Sorter (ARRA,PI,qt)
  4.     Classical_Q_Sorter (ARRA,qt+1,RI)
```

Procedure 1: PARTITION

```
PARTITION(ARRA,PI,RI)
  1.   xt ← ARRA[PI]
  2.   it ← PI-1
  3.   jt ← RI+1
  4.   while (True)
  5.   repeat
  6.     jt ← jt-1
  7.     until (ARRA[jt] <= xt)
  8.   repeat
  9.     it ← it+1
  10.    until (ARRA[it] >= xt)
  11.  Exchange xt↔ ARRA[jt]
  12.  return(jt)
```

3.2 Step by Step Simulation of the Classical Algorithm

Let an array be [9,−3,5,2,6,8,−6,1,3] and obviously not sorted. In the classical Quick Sort last element 3 is considered as a pivot. Each element is compared with pivot and divided into two array where left array is less than pivot and right array is greater than the pivot element. From the divided two array last element is again selected as pivot and again they are divided into sub arrays. This process continues until we get a final sorted array (Fig. 2).

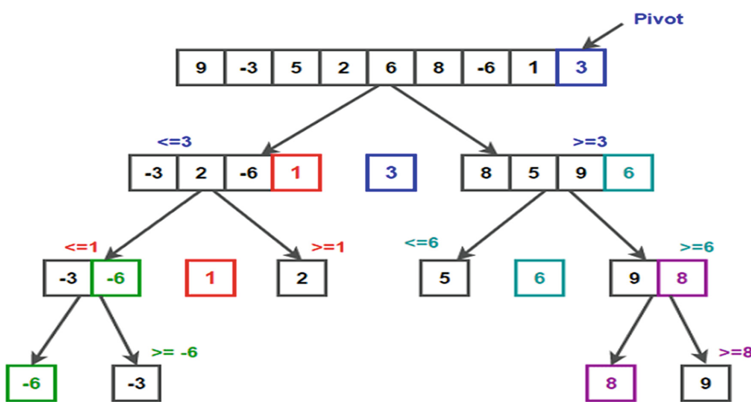

Fig. 2. Step by step simulation of classical Quick Sort.

3.3 Pseudocode of the Proposed Algorithm

This algorithm has three portions. In the first portion, Quick Sort function is called if the size is greater than or equal to 3 otherwise Manual Sort will be called. Then the second portion contains the partition details where each element is compared with the pivot element.

Algorithm 1: QUICK SORT.

```
QUICK_SORT (arr, low, high)
1.  N ← high-low+1
2.  if N ≤ 3
3.      then call procedure 1 MANUAL_SORT (arr, low, high)
4.  Else
5.      do  a←calculate_pivot(arr,low,high);
6.      q← call procedure 2 PARTITION(arr,low,high,a);
7.      QUICK_SORT (arr,low,q);
8.      QUICK_SORT (arr,q+1,high);
```

Procedure 1: MANUAL SORT

```
MANUAL_SORT (arr, low, high)
 1.  N ← high-low+1
 2.  if  n ≤ 1
 3.     then Return;
 4.  if n=2 then
 5.     if arr[low] > arr[high] then
 6.         Exchange arr[low]↔arr[high]
 7.     End if
 8.  End if
 9.  if n=3 then
 10.    if arr[low] > arr[high-1] then
 11.        Exchange  arr[low]↔arr[high-1]
 12.    End if
 13. End if
 14. if arr[low] > arr[high] then
 15.    Exchange arr[low]↔ arr[high]
 16. if arr[high-1] > arr[high]
 17.    Exchange arr[high-1]↔arr[high]
```

Procedure 2: PARTITION

```
PARTITION (arr, low, high, pivot)
 1.  i ← low–1, j ← high+1
 2.  DO while
 3.    IF   TRUE
 4.    while arr[i] < pivot
 5.        then   i++;
 6.    End while
 7.    while arr[j] > pivot
 8.        then   j--;
 9.    End while
 10.    if  i ≥ j
 11.        break;
 12. End DO while
 13. Exchange arr+i ↔ arr+j;
```

3.4 Step by Step Simulation of the Proposed Algorithm

Let, an array be arr [88,77,66,55,44,33,22,11]. Here, the array is not sorted and as the size is greater than 3 it will not call manual sort. So by the pivot selection method this array will be divided into two sub-array of right [88,77,66,55] and left [44,33,22,11]. The max element of the right subarray is 88 and the Min element is 55 whereas the Max element of the left subarray is 44 and the Min element is 11. So the pivot will be the mean of the array. Now each element of two sub-array will be compared with this pivot

element and after applying our proposed Quick Sort algorithm to this array, the results we get are shown through a tree below (Fig. 3, Fig. 4and Fig. 5):

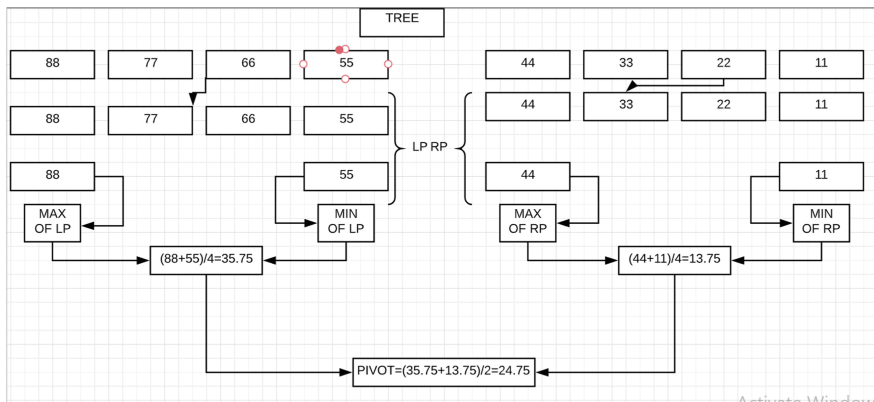

Fig. 3. Tree diagram of PIVOT selection method in Quick Sort.

INITIALLY:

AFTER IMPLIMENTING QUICK SORT:

Fig. 4. Comparison with PIVOT in Quick Sort.

Here, LP = Left Part
RP = Right Part

4 Complexity Analysis

The Best case time complexity of this Quick Sort algorithm is O(*nlogn*), the Worst case time complexity of this algorithm is O(*nlogn*). Analysis of this complexity is described below:

4.1 Time Complexity

Time taken by quicksort, in general, can be written as follows:

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

Here, the first two terms are the two recursive call and the last term is the partition of n elements. The time taken by this algorithm depends on the input of the array and the partition process.

Best Case Analysis
The best-case occurs the algorithm is conducted in such a way that always the median element is selected as the pivot and thus reduces the complexity. The following time is taken for the best case.

$$T(n) = 2T(n/2) + (n)$$

The solution of the above recurrence is O(*nlogn*). It can be solved using Master Theorem. So, the best case of this algorithm is *nlogn* where n is the size the array.

Average Case Analysis
In average case analysis, we need to consider all possible permutations of an array and calculate the time taken by every permutation. The average case is obtained by considering the case when partition puts O(n/9) elements in one part and O(9n/10) elements in other parts. The following time is taken for this:

$$T(n) = T(n/9) + T(9n/10) + O(n)$$

Although the worst-case time complexity of Quick Sort is $O(n^2)$ which is more than many other sorting algorithms Quick Sort can be made efficient by changing the pivot selection method.

Worst Case Analysis
The proposed algorithm gives a better running time than a classical quick sort algorithm. The pivot selection procedure is repeated for each iteration of the quick until the size of the array becomes less than or equal three. In this case, we go for a manual sort where we compare two elements normally. There might be a situation where a worst-case partitioning will be required. When the array will be already sorted or sorted in descending order then worst case partitioning will be needed. Thus mean is calculated and it always comes between extreme values, so, partitioning splits the list into 8-to-2. Thus, the time taken for the proposed algorithm is:

$$T(n) = T(8n/10) + T(2n/10) + cn$$

The recurrence comes to an end when the condition is log10/8(n). The total time taken becomes O(*nlogn*). An 8-to-2 proportional split at every level of recursion making the time taken O(*nlogn*), which is the same as if the split were right down the middle.

Space Complexity
Quick Sort is mainly an in-place sorting algorithm which means it does not need any extra memory. This algorithm works on the 1D array and so it consumes space of n which is basically the size of an array. Thus the space complexity of the full algorithm is O(n) means that the program is running on a linear space algorithm.

5 Experimental Result

In the previous section, we have shown the calculation of the time complexity and space complexity of our proposed algorithm asymptotically. As the new proposed algorithm is not unique, it has some similarities with some sources as all of these are based on the same idea. We have compared our proposed algorithm with these existing algorithms for different input sets in the following sections.

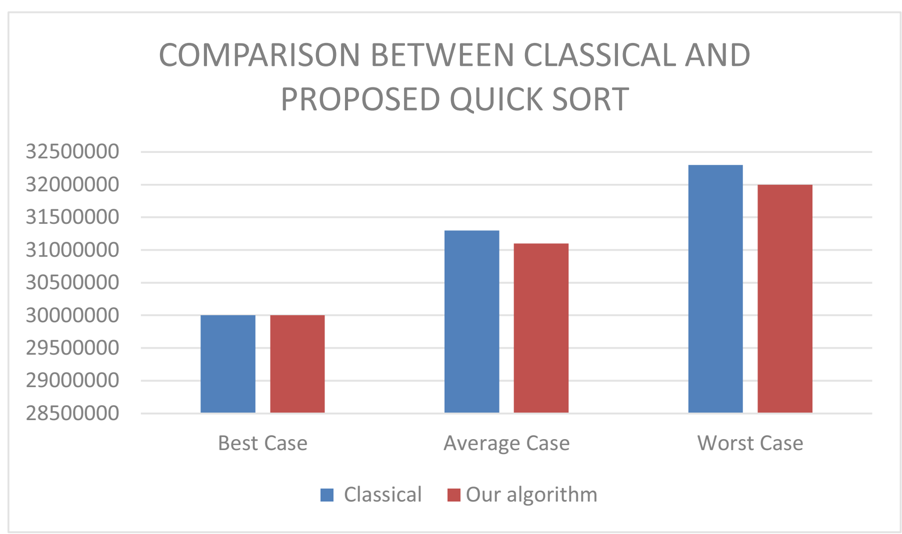

Fig. 5. Runtime comparison chart between the proposed algorithm and the classical algorithm.

Table 1. Runtime(nanosecond) of our proposed algorithm.

Number of input set	Number of elements	Best case	Average case	Worst case
01	25	1000000	1100000	1230000
02	50	2000000	2120000	2230000
03	100	30000000	31000000	32000000

Table 2. Runtime(nanosecond) of the classical algorithm.

Number of input set	Number of elements	Best case	Average case	Worst case
01	25	1000000	1100000	1230000
02	50	2000000	2100000	2220000
03	100	30000000	31000000	32300000

Table 3. Time complexity comparison with previous works.

Algorithm	Best time complexity	Average time complexity	Worst time complexity
Paper[1]	O(*nlogn*)	Ω(*nlogn*)	O(n^2)
Paper[3]	O(*nlogn*)	Ω(*nlogn*)	O(n)
Paper [9]	O(*nlogn*)	Ω(*nlogn*)	O(n)
Proposed Algorithm	O(*nlogn*)	Ω(*nlogn*)	O(*nlogn*)

Here, we present the run time using time function for three different sizes of input sets. For each input set, we have calculated the best case, average case and worst-case execution time in nanoseconds (Table 1 and Table 2):

In Table 3 we represent the comparison of time complexity with other previous works.

6 Conclusion and Future Recommendation

Many researchers researched on the algorithm of Quick Sort to reduce the complexity and make this sorting algorithm more efficient. We presented an algorithm where we tried to use a different method of pivot selection that reduces the comparison and we successfully turned the time complexity to a logarithmic function. We turned it O(*nlogn*) from O(n^2) for the worst time complexity. Obviously, the final choice of implementation will depend on circumstances under which the program will be used. In the future, it seems possible to do further research to calculate pivot in a different way, make the partition process more efficient and handle the worst-case time complexity as perfectly as possible. Moreover, this algorithm is not optimal for large datasets. So in the future, a scope will be created to work with this issue also.

References

1. Latif, A. et al.: Enhancing Quick Sort algorithm using a dynamic pivot selection technique. Wulfenia J. **19**(10), 543–552 (2012)
2. Devi, R., Khemchandani, V.: An efficient quicksort using value based pivot selection an bidirectional partitioning. Int. J. Inf. Sci. App. **3**, 25–30 (2011)
3. Rajput, I.S.: Performance comparison of sequential quick sort and parallel quick sort algorithms. Int. J. Comput. Appl. **57**, 14–22 (2012)
4. Bustos, B.: A dynamic pivot selection technique for similarity search, pp. 105–112 (2008) https://doi.org/10.1109/sisap.2008.12
5. Lakshmi, I.: Performance analysis of four different types of sorting algorithm using different languages. Int. J. Trend Sci. Res. Dev. **2**, 535–541 (2018)
6. Sha, L.: Using simplicity to control complexity. IEEE Softw. **18**(4), 20–28 (2001)
7. Wild, S.: Dual-Pivot quicksort and beyond: analysis of multiway partitioning and its practical potential. Inf. Technol. **60**(3), 173–177 (2018). https://doi.org/10.1515/itit-2018-0012. eISSN 2196-7032, ISSN 1611-2776

8. Aumüller, M., Dietzfelbinger, M., Klaue, P.: How good is multi-pivot quicksort? ACM Trans. Alg. **13**, Article no. 8 (2016)
9. Aumüller, M., Dietzfelbinger, M.: Optimal partitioning for dual-pivot quicksort. ACM Trans. Alg., vol. 12, Article no 18 (Association for Computing Machinery) (2015)
10. Kushagra, S., López-Ortiz, A.: Multi-pivot quicksort: theory and experiments (2013)
11. Faujdar, N., Prakash, S., Professor, G.: Performance evaluation of merge and quick sort using GPU computing with CUDA. Int. J. Appl. Eng. Res. vol. 10 (2015)
12. Abdulrahman Hamed Almutairi, B., Helal Alruwaili, A., Hamed Almutairi, A.: Improving of quicksort algorithm performance by sequential thread or parallel algorithms (2012)
13. Bahig, Hazem M.: Complexity analysis and performance of double hashing sort algorithm. J. Egyptian Math. Soc. **27**(1), 1–12 (2019). https://doi.org/10.1186/s42787-019-0004-2
14. Jadoon, S., Salman, F.S., Rehman, S.: Design and analysis of optimized selection sort algorithm. Int. J. Elect. Comput. Sci. IJECS-IJENS **11**(02), 19–24 (2011)
15. Rathi, N.: QSort-Dynamic pivot in original quick sort. Int. J. Adv. Res. Devel. (2018)