International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 08 Issue: 12 | Dec 2021 www.irjet.net p-ISSN: 2395-0072

Divide and Conquer Sorting Techniques

Ashutosh Gautam[1], Agrasth Naman[2]

[1]*Student, Dept. of Computer Engineering, KIIT University, Odisha, India*
[2]*Student, Dept. of Computer Engineering, KIIT University, Odisha, India*

Abstract - *This paper shows the combination between the two most popular divide and conquer sorting algorithms i.e. quick and mergeSort. Both of these algorithms have their own merits and demerits. Each of these two algorithms attempts to sort the data of the problem in a distinct format. This paper makes an attempt to present a detailed comparative study of how the two algorithms work and tries to show the difference between the performance of the two on the basis of both space and time to reach a final interference.*

Key Words: Divide and conquer, MergeSort, QuickSort Comparisons, Time Complexity, Space Complexity.

1. INTRODUCTION

A sorting algorithm is a method of rearranging the data or items of the list in a specific order; the order can be escalating or DE-escalating. Sorting algorithms can be divided into two major categories:-Comparisons based sorting technique and non comparisons based sorting technique.[1]

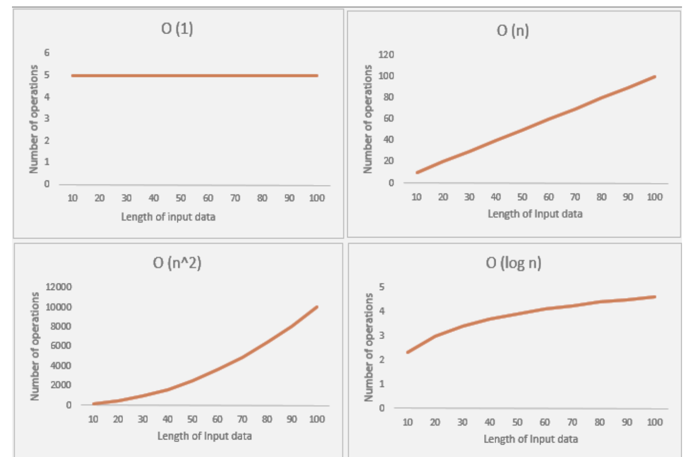
In the comparison based sorting technique, we use a block to compare two data of blocks and then swap or copy those elements if required and it continues executing until and unless the whole list or array is arranged in a certain order.

Non comparison based sorting can only be used in some particular cases and it involves sorting the element of a list or array on the basis of their internal character of the value.

Quick and merge sort both use comparison based sorting techniques as well as employ a common divide and conquer paradigm. Divide and conquer algorithms involve solving a problem by using recursion as the main task is divided into smaller sub-tasks and finally combine the solution of the sub-task to solve the given original task.

Comparison of performance of sorting Algorithms involves the two main parameters:-

Time Complexity represents the number of times an operation statement is executed. It's not the actual time required to execute an operation, it also depends upon factors like processing power, programming language, operating software , etc.[2]



2. PROCEDURE OF ALGORITHMS

Merge Sort:

It takes an input list and divides it until we are left with a bunch of sub-list of size one and that are trivially sorted, then the merging process begins.

It sequentially compares the elements of two sub-lists together to form sorted sublists of size 2, then repeats the process to form a sub-list of size 4 then size 8, and so on and this happens until it has just one sorted sublist with the same size of the input list. At this point, the list is sorted.

It takes O(log N) operations for a merge sort to divide an input list of n elements into n sublists of one element. Then it takes O(n) operations to merge the sub-list together thus it has a time complexity of O(n log n).

It is a stable algorithm and it can be further optimized in practice by merging sub-list in parallel with one another.

The biggest drawback to merge sort is that an auxiliary space of O(n) is required during the merging process.[3]

Merge Sort is based on the divide and conquer technique.

Let's assume MergeSort( ) is a function that takes the input array, then calls itself for its two halves, then merges the sorted halves.

And another function Merge is used to merge the two halves.

- For MergeSort(list[],l ,m)

L, R, M as left, right, medium index respectively

if r > l

m = l + (r-l)/2

MergeSort(list,l,m)

MergeSort(list,m+1,r)

Merge(list,l,r,m)

- For Merge(list,l,r,m)

Size1=m-l+1

Size2=r-m

Declare 2 array Arr1[] & Arr2[]

For i=0 to Size1

Arr1[i]=list[l+i]

For j=0 to Size2

Arr2[j]=list[m+1+j]

Declare i=0 j=0 k=l

While i<Size1 and j<Size2

if Arr1[i] <=Arr2[j]

List[k]=Arr[i] i++

Else list[k] = Arr2[j] j++

k++

While i < Size1

List[k]=Arr1[i]

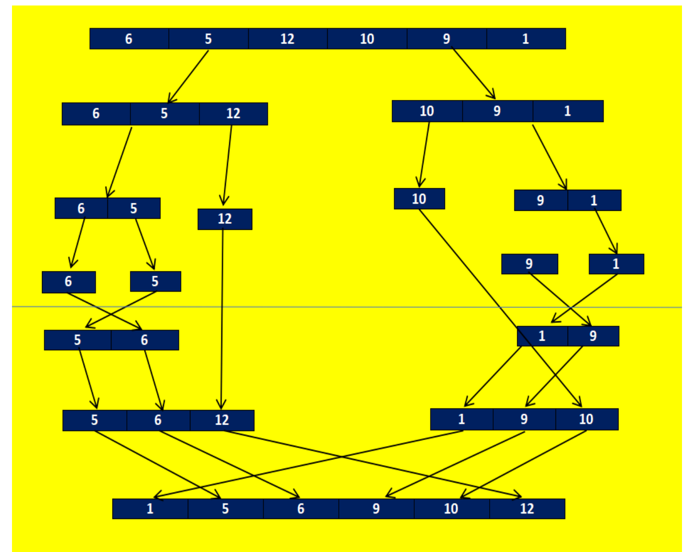
i++ k++

While j < Size2

List[k]=Arr2[j]

j++ k++

Exit.



Few runtime data for merge sort according to the number of input elements. [4]

Number of elements	Running time (in ms)
10000	728
20000	1509
30000	2272

Quick Sort:-

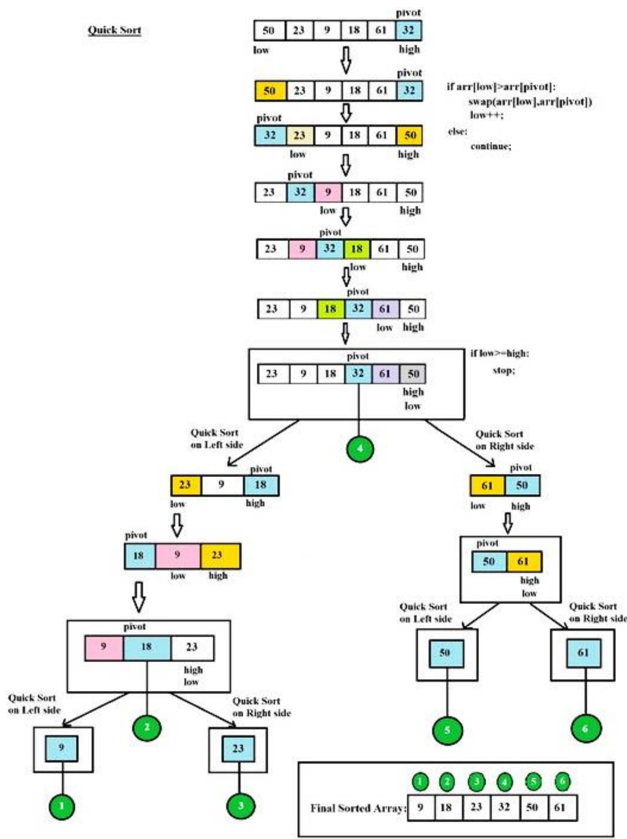
It first picks an element from the input list, called the pivot, all elements less than the pivot are placed before it and all the elements greater than it is placed after it. Once this step is completed, then the pivot is in its final position and the input list has been partitioned into two sub-lists elements less than the pivot, i.e. elements less than the pivot and elements greater than the pivot.[5]

It then recursively applies the same step to each sub-list until it has a sub-list of at most one element which is trivially sorted. Once the recursion is finished the list is sorted on average as long as the chosen pivot divides the input list into recently sized pieces. It takes log n recursive calls to reach a list size of one. For each recursive call, it takes n operations to place the other elements around the pivot. Therefore it has an average time complexity of O(N LOG N).[6] At the same time, its worst-case time complexity is O(N^2). It happens when the chosen pivot is always in minimum or maximum and they actually don't partition the list at all.

This technique is used in almost sorted data.

However, the probability of occurring on a large random input is extremely small. So we generally consider the worst case the run time of quicksort to be of n log n.

In practice quicksort tends to be faster than merge sort; this is because it uses log n stack space as it recursively partitioning the input list and it is usually not implemented as a stable sorting algorithm.



Let us assume QuickSort is a function that is used to place elements before and after the targeted pivot. The function partition takes the element as a pivot element at the correct position in the sorted array.

- For QuickSort

If lowerIndex < higherIndex

Pivot=partition(arr,lowerIndex,higherIndex)

QuickSort(arr,lowerIndex,pivot-1)

QuickSort(arr,pivot+1,higherIndex)

- For partition

Pivot = array[high]

i = lowerIndex-1

For j = lowerIndex to a=higherIndex-1

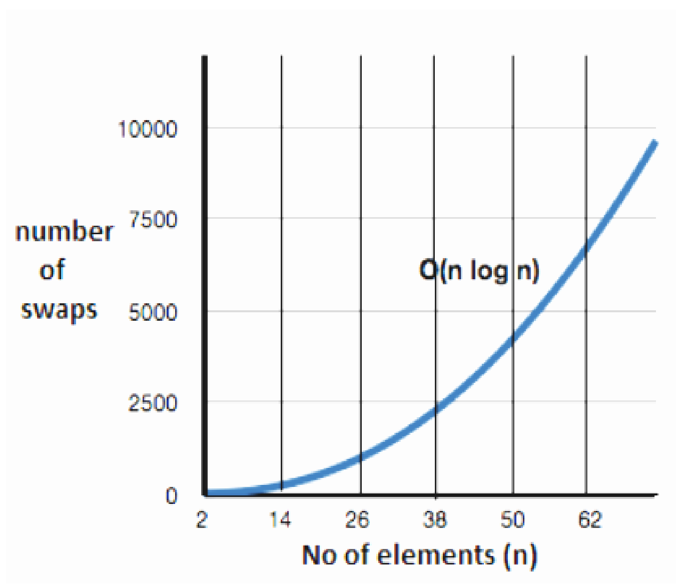
If array[j]<pivot

i++ and swap array[i]and array[j]

Swap array[i+1] and array[higherIndex]

Return i + 1

Number of elements	Running time (in ms)
10000	489
20000	1084
30000	1648



Time required for quick sort for a list of elements is: Let the size of array is denoted by SIZE,

T(SIZE) = T(P) + T(SIZE - P - 1) + O(SIZE)

where P is the count of elements less than the pivot.

Worst Case:

The worst case happens when the division method always picks the maximum or minimum element as a pivot. If we observe our division strategy, where 1st is always picked as pivot, then the worst case would occur when the list is already arranged in escalating or de-escalating order. Hence, the recurrence relation becomes:

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

T(SIZE) = T(SIZE - 1) +SIZE

T(SIZE - 1) = T(SIZE -2) + SIZE -1

T(SIZE) = T(SIZE - 2) + SIZE - 1 + SIZE

T(SIZE) = T(SIZE - 3) + SIZE -2 + SIZE -1 + SIZE

T(SIZE) = 1 + 2 + 3 + ...+ SIZE

T(SIZE) = (SIZE (SIZE + 1))/2

Here, the time complexity is O(n^2), where n is SIZE.

Best case: The Best case occurs when the middle element is chosen as a pivot for the partition process. Thus the recurrence relation becomes:

T(SIZE) will be 0 if SIZE is equal to 1, otherwise:

T(SIZE) = 2T (SIZE/2) + O(SIZE).

T(SIZE) = 2T (SIZE /2) +SIZE

T(SIZE /2) = 2T (SIZE /4) + SIZE /2

T(SIZE) = 2{2T(SIZE /4) + SIZE /2}+SIZE

T(SIZE) = 2^P T(SIZE /2^P) + k SIZE

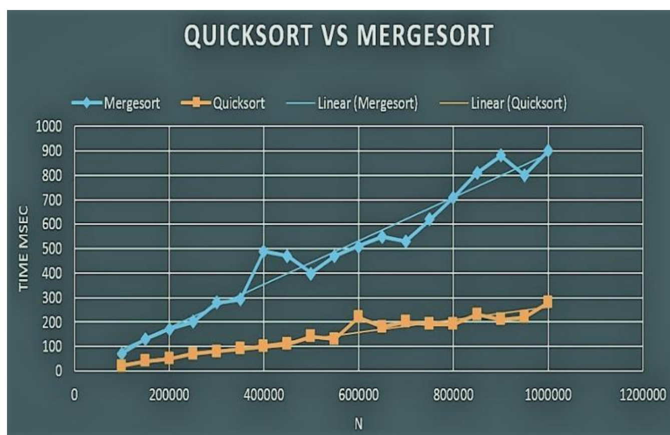
$2^P = n$, $k = \log_2 n$, $T(1) = 0$, where n is SIZE

T(SIZE) = SIZE T(1) + SIZE $\log_2$ SIZE

T(SIZE) = SIZE .0 + SIZE $\log_2$SIZE

T(SIZE) = SIZE $\log_2$ SIZE

Here, the time complexity is O(nlogn), where n is SIZE.



3. CONCLUSIONS

In this study a comparison of performance has been done between the two most popular divide and conquer sorting techniques with the parameter of space and time complexity. Quicksort was found to be very efficient on smaller datasets while merge sort would be recommended on larger datasets. Although merge sort is faster, its requirement of extra memory space of O(n) is very inefficient in comparison to quicksort where the space requirement is O(log n). So, on smaller data sets (less than 400 elements) it is preferable to use quicksort while merge sort should be the first choice when dealing with relatively larger datasets. QuickSort is recommended to all data sizes if cache locality needs to be used. This is believed to be very useful for future programmers as it gives more clarity of choice when they face a dilemma on when and where to use each of these two sorting techniques.

ACKNOWLEDGEMENT

The authors would like to thank Dr.Santosh Kumar Baliarsingh (Assistant Professor in the School of Computer Engineering, KIIT, Deemed to be University, Bhubaneswar) for his valuable insights and suggestions on the paper.

REFERENCES

1) J. Phongsai, "Research paper on Sorting Algorithm," 2009.

2) D. Knuth, in The art of programming sorting and searching,1988.

3) Aditya, DM., Deepak, G. Selection of Best Sorting Algorithm. International Journal of Intelligent Information Processing.

4) Jules RT, Algorithms and Complexity Theory, Durban (2010)

5) J.-p. T. T. M. Hill, An introduction to Data Structure with the application.

6) Shuang, C., Shunning, J., Bingsheng, H., Xuenyan, T. A Study of Sorting Algorithms on Approximate Memory. San Francisco. 2016

BIOGRAPHIES

Ashutosh Gautam is a sophomore at Kalinga Institute of Industrial technology, Bhubaneswar. He is pursuing a bachelor's degree in technology in the computer science branch.

Agrasth Naman is a sophomore at Kalinga Institute of Industrial technology, Bhubaneswar. He is pursuing a bachelor's degree in technology in the computer science branch.