ISSN: 2831 - 6789

Research Article

Journal of Robotics and Automation Research

Establishing Pertinence between Sorting Algorithms Prevailing in *n log(n)* time

Manan Roy Choudhury[1], Anurag Dutta[2*]

Undergraduate, Computer Science and Engineering, Government College of Engineering and Textile Technology, Serampore-712201, India

[*]Corresponding author
Anurag Dutta, Undergraduate, Computer Science and Engineering, Government College of Engineering and Textile Technology, Serampore-712201, India

Submitted: 28 Jul 2022; **Accepted**: 02 Aug 2022; **Published**: 16 Aug 2022

Citation: *Choudhury, M.R., Dutta, A. (2022). Establishing Pertinence between Sorting Algorithms Prevailing in n log(n) time. J Robot Auto Res 3(2), 220-226.*

Abstract
Data is the new fuel. With the expansion of global technology, the increasing living standards, and modernization, data values have caught great height. Nowadays, nearly all top MNCs feed on data. Now, storing all this data is a prime concern for all of them, which is relieved by the Data Structures, the systematic way of storing data. Now, once these data are stored and charged in secure vaults, it's time to utilize them most efficiently. Now, many operations need to be performed on these massive chunks of data, like searching, sorting, inserting, deleting, merging, and so more. In this paper, we would be comparing all the major sorting algorithms, that have prevailed to date. Further, work has been done and inequality in the dimension of time between the three Sorting algorithms, operational in n log(n) time, Merge, Quick, and Heap, that have been discussed in the paper have been proposed.

Keywords: Sorting, Heap Sort, Merge Sort, Quick Sort, Array, Asymptotic Notation, Time Complexity.

Introduction

Sorting is one of the most important but basic operations that may be enacted on any data structure. It involves arranging the data in monotonic order of its magnitude [1]. Sorting algorithms have got a lot to flaunt, it's the well-balanced and cunning nature with spikes of intelligence, its efficiency, and not only that, even some searching algorithms like binary search, interpolation search need sorting algorithms to drop them in action. The orders most often used are numerical order and lexicographic order, and either upward or downward [2,3,4].

Any sorting algorithm's output must meet two formal requirements:

- The output is in either increasing or decreasing order (each element is the same size as the one before it, in the order specified).
- The output is a monotonic arrangement of the initial array (a reordering of the input while keeping all of the original elements).
- The input data is stored in a data structure that allows random access rather than sequential access for maximum efficiency.

The sorting challenge have garnered a large deal of research since the dawn of computing, probably due to the difficulty of addressing it effectively despite its basic, common expression. Betty Holberton, who collaborated on Enigma machine and UNIVAC, was one of the early creators of sorting algorithms about 1951 [5]. Bubble sort has been studied since 1956. Asymptotically optimum algorithms have been recognized since the mid-twentieth century; new algorithms are continually being developed, with the extensively used Timsort dated from 2002 and the library sort from 2006.

The necessity of $\Omega(n \log n)$ comparisons in comparison sorting algorithms is fundamental [6]. Algorithms that aren't focused on comparisons, such as counting sort, often perform better. The abundance of methodologies for the conundrum offers a comprehensive guide to a diverse array of fundamental heuristic notions, such as big O notation, divide and conquer algorithms, data structures such as heaps and binary trees, randomized algorithms, best, worst, and average-case analysis, time-space tradeoffs, and upper and lower bounds [7,8].
Sorting algorithms are classed as follows:

- Complexity of computation: In the perspective of list size, the best, worst, and average case scenarios exist. The good behavior of typical serial sorting algorithms is $O(n \log n)$, while the bad behaviour is $O(n^2)$. The ideal behaviour for a serial sort is O(n), although in most cases, this is not attainable. The best parallel sorting algorithm is O(log n).
- Memory consumption: Some sorting algorithms, in particular, are "in-place." Beyond the entries being sorted, an in-place sort requires only O(1) memory; nonetheless, O(log

n) supplementary cognition is frequently considered "in-place" [9].

- Recursive in nature: Some algorithms are recursive or non-recursive, where recursion means that the function will call itself indefinitely such that to attain its final value.
- Cohesion: stable sorting algorithms maintain records with equal attributes in just the same order. Whether they are a comparison category or not. A comparison sort compares two components with a comparison operator to analyze the data.
- Generalized Approach: Insertion, exchange, selection, merging, and other general methods Bubble sort and quick-sort are examples of exchange sorts. Cycle sort and heapsort are two types of selection sorts. The algorithm's serial or parallel nature. Our paper primarily focuses almost entirely on serial algorithms and assumes that they are used in serial mode.
- Adaptive Nature : Whether the array is pre-sorted, still the source has an impact on the run time. Adaptive algorithms are those that incorporate everything into consideration.
- Continuous: An interactive algorithm, such as Insertion Sort, can semblance a continuous transmission of bits [10].

Merge sort is a broad sense, resemblance sorting algorithm developed in computer science. The plurality of implementations build a sustainable sort, essentially implies that perhaps the order of identical bits in the source and load is the same. John von Neumann devised merge sort in 1945 as a divide-and-conquer algorithm. Goldstine and von Neumann published a study in 1948 that included a comprehensive description and assessment of underside merge sort [11,12].

Heapsort is a resemblance sorting method in computer science. Heapsort is similar to selection sort in that it separates its input together into sorted and an unsorted region, then progressively decreases the unsorted part by taking the pivot element from it and putting it into the sorted portion. Unlike selection sort, heapsort does not waste time scanning the unsorted region in linear time; instead, heap sort keeps the unsorted region in a heap data structure to identify the largest element in each step more rapidly.

Quicksort is a sorting algorithm that works in-place. It was developed by British computer scientist Tony Hoare in 1959 and publicized in 1961, but it is still a prominent sorting algorithm. It can be marginally quicker than merge sort and 2 to 3 times quicker than heapsort when properly implemented. Quick Sort follows divide and conquer technique. It works by selecting a 'pivot' component from the arrays and partitioning the remainder into two sub-arrays based on whether they are below than or larger than the pivot. As a consequence, it's also known as partition-exchange sort [13]. The sub-arrays are then recursively sorted. This could be done in place, with only a small proportion of total RAM required for categorizing.

Although this is slightly slower in practice on most processors than a well-implemented quicksort, it has a better worst-case O(n log n) latency.

The merit of the paper covers the proposition of a new inequation relating to the time elapsed by Sorting algorithms operational in *n log(n)* time - Merge, Quick, and Heap Sort. To enact more on our inequation, both the Average and the Worst-Case Scenario have been considered.

Merge Sort

Merge sort is a very good sorting technique as it follows the divide and conquer algorithm [14].

Let we have been given a set of unsorted elements in a list (data structure with language independence), such that

$$L(n)=\{\alpha_0,\alpha_1,\alpha_2,\ldots,\alpha_{n-1}\}$$

Under this algorithm the list is divided into equally sized subparts and merged step by step in a recursive manner to bring it to a sorted format [8]. It is often referred to as the best sorting technique when we are required to sort a linked list.

The Pseudocode for this algorithm will be

```
function merge (list_of_elements, low_index, mid_index, high_index)
size_vault1 ← mid_index - low_index + 1
size_vault2 ← high_index - (mid_index + 1) + 1
vault1[size_vault1]
vault2[size_vault2]
for i = 0 to i = size_vault1 - 1
vault1[i] = list_of_elements[low_index + i]
end for
for i = 0 to i = size_vault2 - 1
vault2[i] = list_of_elements[mid_index + 1 + i];
end for
i, j ← 0, 0
k ← low_index
while i < size_vault1 and j < size_vault2
 if vault1[i] > vault2[j]
 list_of_elements[k] = vault2[j]
increment j, k
else
list_of_elements[k] = vault1[i];
increment i, k
end if
end while
while j < size_vault2
 list_of_elements[k] = vault2[j];
increment j, k
end while
while i < size_vault1
list_of_elements[k] = vault1[i];
increment i, k
end while
end
function merge_sort (list_of_elements, low_index, high_index)
if low_index < high_index
```

```
mid_index ← low_index + (high_index - low_index)
merge_sort(list_of_elements, low_index, mid_index)
merge_sort(list_of_elements, mid_index + 1, high_index)
merge (list_of_elements, low_index, mid_index, high_index) end if
end
```

The Complexity in the dimensions of time for this Sorting Algorithm for worst cases will be $\varphi(n\ log_2 n)$.
The Complexity in the dimensions of time for this Sorting Algorithm for average cases will be $\varphi(n\ log_2 n)$.
The Complexity in the dimensions of time for this Sorting Algorithm for best cases will be $\varphi(n\ log_2 n)$.

where $\varphi(.)$ is the appropriate Asymptotic Notation.

Quick Sort

Quicksort is a very good sorting technique as it follows the divide and conquer algorithm [15].
Let we have been given a set of unsorted elements in a list (data structure with language independence), such that

$L(n)=\{\alpha_0,\alpha_1,\alpha_2,\ldots,\alpha_{n-1}\}$

Under this algorithm, we choose an element as a pivot and we create a partition of array revolving around that pivot. By repeating this technique for each partition, we get our array sorted depending on the position of the pivot we can apply quick sort in different ways

- Taking the first or last element as a pivot
- Taking median element as pivot.

The Pseudocode for this algorithm will be

```
function partition (left, right, pivot)
leftPointer ← left
rightPointer ← right - 1 while True do
while list_of_elements[++leftPointer] < pivot do
end while
while rightPointer > 0 and list_of_elements[--rightPointer] > pivot do
end while
if leftPointer ≥ rightPointer
break
else
swap leftPointer, rightPointer
end if
end while
swap leftPointer, right
return leftPointer
end
function quicksort(left, right)
if right ≤ left
return
else
pivot ← list_of_elements [right]
part ← partition(left, right, pivot)
quickSort(left, part - 1)
quickSort(partition + 1,right)
end if
end
```

The Complexity in the dimensions of time for this Sorting Algorithm for worst cases will be $\varphi(n^2)$.
The Complexity in the dimensions of time for this Sorting Algorithm for average cases will be $\varphi(n\ log_2 n)$.
The Complexity in the dimensions of time for this Sorting Algorithm for best cases will be $\varphi(n\ log_2 n)$.

where $\varphi(.)$ is the appropriate Asymptotic Notation.

Heap Sort

Heap sort is a comparison-based sorting technique based on Binary Heap data structure
Let we have been given a set of unsorted elements in a list (data structure with language independence), such that

$L(n)=\{\alpha_0,\alpha_1,\alpha_2,\ldots,\alpha_{n-1}\}$

Heapsort can be thought of as an improved selection sort [16]. Like selection sort, Heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region.

The Pseudocode for this algorithm will be

```
function heapify
max ← i
left_child ← 2i + 1
right_child ← 2i + 2
if leftchild ≤ n and A[i] < A[leftchild]
max ← leftchild
else
max ← i
end if
if right_child ≤ n and list_of_elements[max] > list_of_elements[right_child]

 max ← right_child
 end if
 if max not equal i
   swap(list_of_elements[i], list_of_elements[max])
 heapify(list_of_elements, n, max)
end if
end
function Heapsort
n ← length(list_of_elements)
for i from n/2 to 1
Heapify(list_of_elements, n ,i)
 for i from n to 2
exchange list_of_elements [1] with list_of_elements [i]
```

```
list_of_elements.heapsize   =   list_of_ele-
ments.heapsize - 1
 Heapify(list_of_elements, i, 0)
end
```

The Complexity in the dimensions of time for this Sorting Algorithm for worst cases will be $\varphi(n\ log_2 n)$.
The Complexity in the dimensions of time for this Sorting Algorithm for average cases will be $\varphi(n\ log_2 n)$.
The Complexity in the dimensions of time for this Sorting Algorithm for best cases will be $\varphi(n\ log_2 n)$.

where $\varphi(.)$ is the appropriate Asymptotic Notation.

Worst Case Scenario

Every program, perhaps once in it's run time, faces the difficulty to go the hurdles and difficulties which stick to their maximum at that point of time, such a case is termed as Worst Case. We have plotted pie chart for 10 sets of data, with 10 trials for each set, involving 3 types of Merge, Quick, and Heap Sort taking the time taken in terms of 10^{-9} seconds or nanoseconds by them as a whole of 100%. and the result was quite promising. The data set used was arranged in descending order in terms of it's magnitude for the case for Merge and Heap Sort, and in ascending order for Quick Sort, and the algorithms we designed arranged them in ascending order of their magnitude.

Table 1: Time taken in nano - seconds (averaged for 10 tests) for MQH Sort.

Number of Elements	*Merge Sort*	*Quick Sort*	*Heap Sort*
1000	98400	784100	395400
10000	199800	3063000	2367000
100000	272000	10454300	299100
1000000	0	18120400	544000
10000000	401800	28290500	796700
100000000	598900	35658800	808400
1000000000	296800	50567100	799400
10000000000	799000	70179400	1166200
100000000000	598800	88498900	1196100
1000000000000	697100	108973400	878700

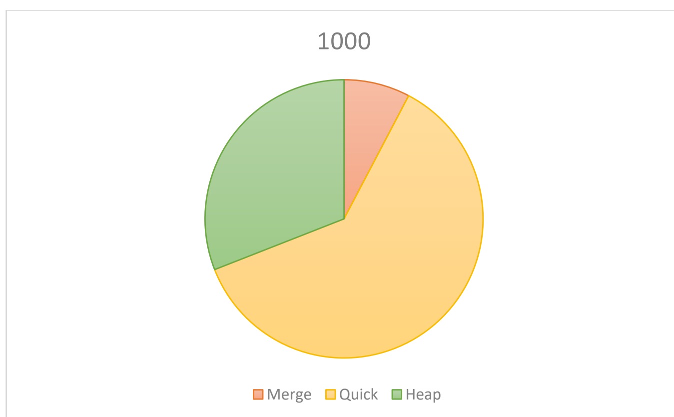

Figure 1: For a Data set of 1000 entries

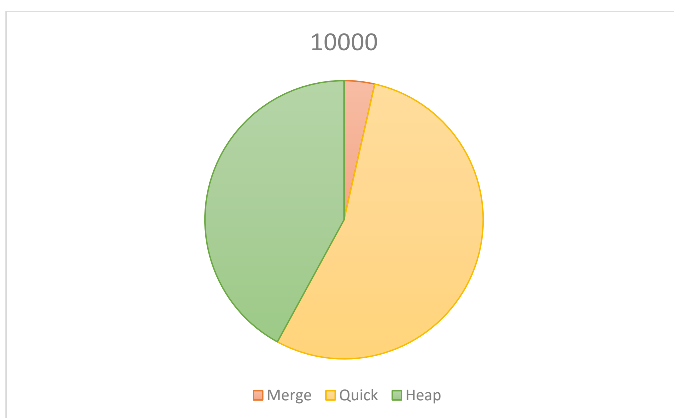

Figure 2: For a Data set of 10000 entries

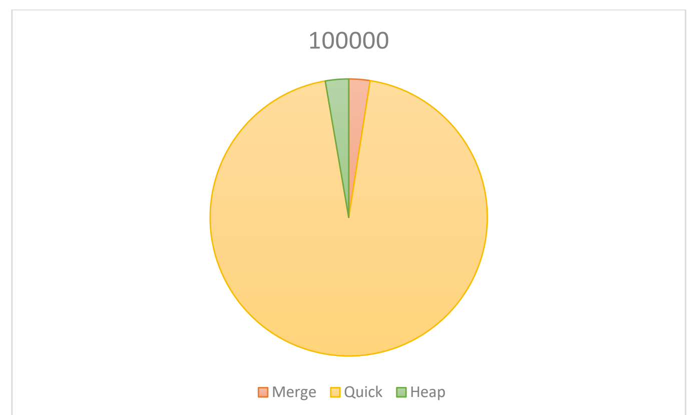

Figure 3: For a Data set of 100000 entries

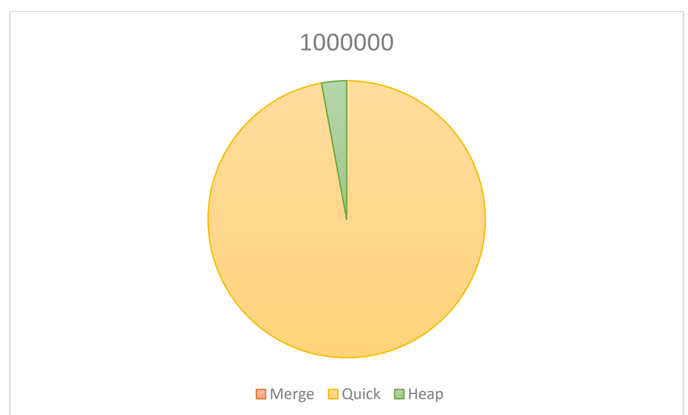

Figure 4: For a Data set of 1000000 entries

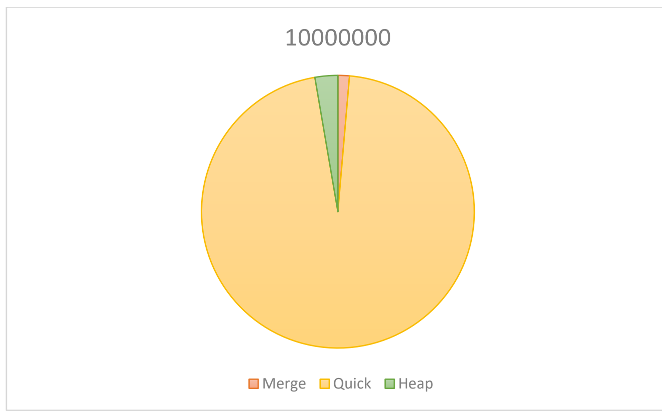

Figure 5: For a Data set of 10000000 entries

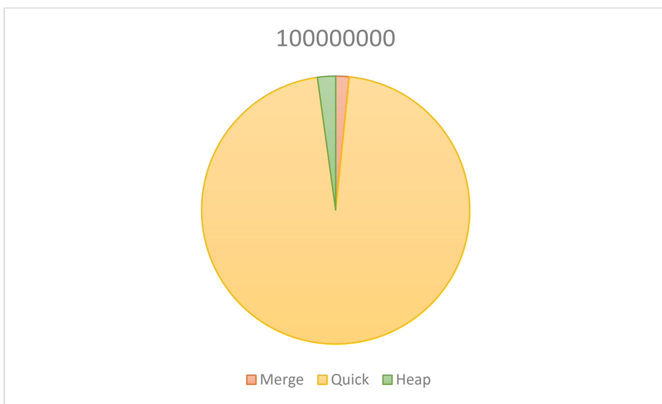

Figure 6: For a Data set of 100000000 entries

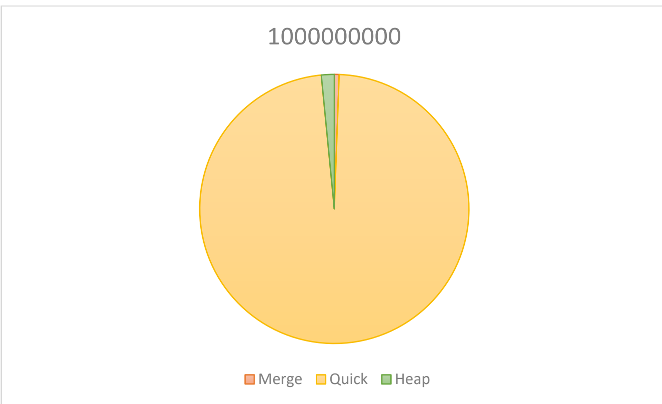

Figure 7: For a Data set of 1000000000 entries

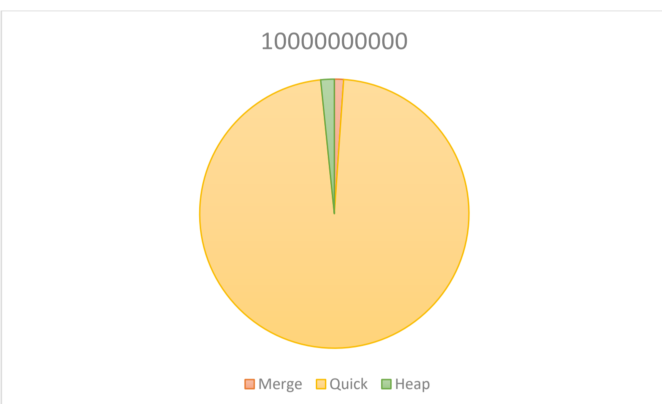

Figure 8: For a Data set of 10000000000 entries

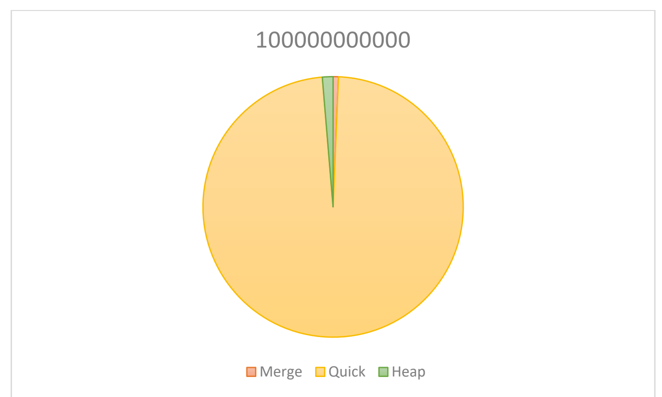

Figure 9: For a Data set of 100000000000 entries

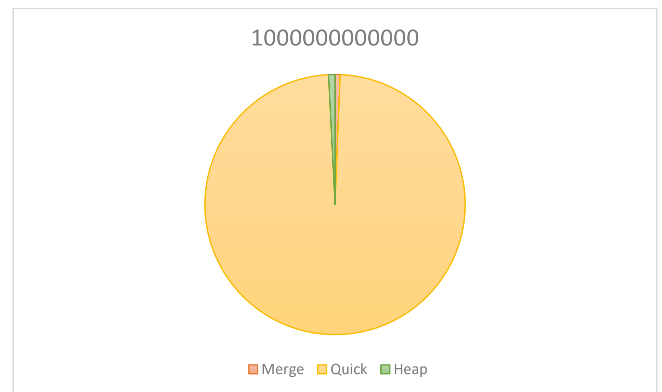

Figure 10: For a Data set of 1000000000000 entries

Now, we will undertake a detailed statistical assay on the exact execution time of the three sorting algorithms i.e., Merge Sort, Heap Sort and Quick Sort. Taking in consideration the execution time of these three sorting algorithms, the computer architecture on which we ran these algorithms becomes one of the main factors to consider. To be precise we have used Harvard architecture to carry our tests runs of these algorithms. The precise details of the architecture we used is given below :

o Device Specifications

Processor Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz, 1.50 GHz Installed
RAM 16.0 GB (15.8 GB usable)
System type 64-bit operating system, x64-based processor

o Windows Specification

Edition Windows 11 Home Single Language
Version 21H2
OS build 22000.613
ExperienceWindows Feature Experience Pack 1000.22000.613.0
The System is having a Harvard Architecture.

Now we will analyse the execution time of Quick sort :
When one of the subsists returned by the partitioning procedure is of size n-1, the partition is the most imbalanced. This could happen if the pivot is the least or largest element in the list, or if all the items are equal in some implementations.

If this applies throughout every partition, subsequent recursive operation would process a listing that really is unit size smaller than the prior list. As a result, before we reach a list of size 1, we can make n-1 nested calls. The call tree is therefore a linear sequence of n-1 nested calls. So, the execution time turns out to be

$$\sum_{i=0}^{n}(n-i)+O(1)=\frac{n(n+1)}{2}+O(1).$$

For Merge sort, to find the middle of any subarray, we use a single-step using an one-step operation.
An *O(n)* execution time of *O(n)* will be required to integrate the subarrays created by partitioning the initial array of *n* elements. As a result, the overall time for the Merge sort function will be $O(1)n(\log n + 1)$

Binary heaps are predicated on comprehensive binary trees; the bottom level will have n/2 nodes, the second category will always have n/4 nodes, and so on. We cut the number of nodes in half whenever we advance a threshold.
When we add everything up, we get:

$$\frac{n}{4}+2\times\frac{n}{8}+3\times\frac{n}{16}+\cdots$$

This can also be expressed as a summarization: $\sum_{i\in\mathbb{N}} i\times\frac{n}{2^{i+1}}$

This summation turns out to be $1-\left(\frac{1}{2}\right)^{(floor(log_2 n)+1)}$.

Now, our aim is to evacuate the element α_i to its original location. To get back to its original location, we'll have to look in as many areas as possible. Now, once it has been dumped in its original location, we will go on to the next element of and in order to evacuate it to its proper location, we must search at least n-1 locations, with the number of locations to be searched varying from *n*-3,*n*-4,*n*-5,*n*-6,...,1 [17].
So, total time

$$\mathcal{T}(n)=log_2 n+log_2(n-1)+log_2(n-2)+\cdots+log_2 1$$

$$\mathcal{T}(n)=\sum_{i=0}^{n-1}log_2(n-i)$$

$$=log_2\left(\prod_{i=0}^{n-1}(n-i)\right)=log_2(n!)$$

If we consider the Stirling's Approximation,

$$\mathcal{T}(n)|_{min}=log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\left(\frac{1}{12n+1}\right)}\right)$$

$$\Rightarrow\mathcal{T}(n)|_{min}=log_2\left(\sqrt{2\pi}n^{\left(n+\frac{1}{2}\right)}\right)+log_2\left(e^{\left(\frac{1}{12n+1}\right)-n}\right)$$

$$\Rightarrow\mathcal{T}(n)|_{min}\approx\left(n+\frac{1}{2}\right)log_2(n)+n\approx n\,log_2(n)$$

Thus, the total execution time of Heap sort is

$$n\,log_2(n)\left(1-\left(\frac{1}{2}\right)^{(floor(\log_2 n)+1)}\right).$$

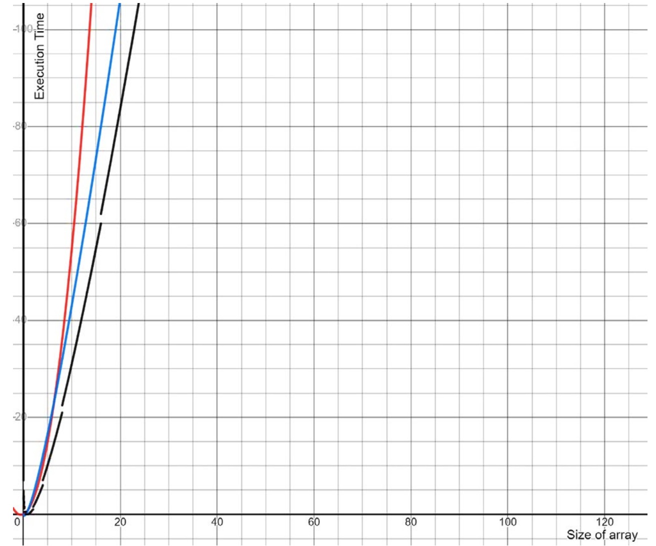

Figure 11: Graphical representation of comparative analysis of run time of Merge, Quick and Heap sort.

Here, the blue curve represents the run time of Merge sort, the red curve represents the run time of Quick sort and the black curve represents the run time of Heap sort respectively.

Conclusion

From the study, we have conducted in this paper, we can conclude that, in each of trials, irrespective of Architecture and Specification of the computational device. It is clear from the figure 10 that the worst case run time taken by Heap sort to sort certain elements in a given array is less than the worst case run time taken by both Quick sort and Merge sort.

$T(Q) \geq T(M) + T(H)$ & $T(M) < T(H)$
$\ni Q = \text{Quick Sort}, M = \text{Merge Sort}, H = \text{Heap Sort}$

where,
T(ξ) is the time taken in terms of nanoseconds by that specific sorting algorithm ∀ξ∈ Comparison Sorting Techniques and

$$0\% \leq \left(\frac{\delta\varepsilon}{\varepsilon}\right)\times 100 \leq 5\%,\ \varepsilon, \text{ being the tolerance limit.}$$

This error / tolerance limit is due to the variation in processing speed and time due to various architectural aspects and physical aspects. Some of the aspects that may result in increasing the tolerance limit are:
System Temperature (θ_s): If $\theta_s > \theta_0$, overheating of computation system occurs, that results in decrease of processing speed, and hence claims more time.

$$\frac{\delta\varepsilon}{\varepsilon}\propto\theta_s-\theta_0$$

where, θ_0=Threshold Temperature

Network Stability: This issue is more taken in account if the computation is done online. The instability in network may induce more time being claimed.
Power Stability: If the machine is undergoing a sudden surge or

decline in power, the performance may be hindered, resulting in claim of more time.

The data sets that we have obtained in this study are generated by system working on Harvard Architecture,though the inequation developed is irrespective of the architecture.

References

1. Cormen, T. H. (2001). Charles E Leiserson Ronald L Rivest, Clifford Stein. Introduction to algorithms.
2. Williams Jr, L. F. (1976, April). A modification to the half-interval search (binary search) method. In Proceedings of the 14th annual Southeast regional conference (pp. 95-101).
3. Gonnet, G. H., Rogers, L. D., & Alan George, J. (1980). An algorithmic and complexity analysis of interpolation search. Acta Informatica, 13(1), 39-52.
4. Perl, Y., & Reingold, E. M. (1977). Understanding the complexity of interpolation search. Information Processing Letters, 6(6), 219-222.
5. Design, A. (2005). Jon Kleinberg, Eva Tardos.
6. Harel, D. (1987). Algorithmics: The Spirit of Computing, Springer.
7. Williams, J. W. J. (1964). Algorithm 232: heapsort. Commun. ACM, 7, 347-348.
8. Wilf, H. S. (2002). Algorithms and complexity. AK Peters/ CRC Press.
9. Malhotra, D., & Chug, A. (2021). A modified label propagation algorithm for community detection in attributed networks. International Journal of Information Management Data Insights, 1(2), 100030.
10. Nair, R. S., Agrawal, R., Domnic, S., & Kumar, A. (2021). Image mining applications for underwater environment management-A review and research agenda. International Journal of Information Management Data Insights, 1(2), 100023. ISSN 2667-0968.
11. Rajendran, D. P. D., & Sundarraj, R. P. (2021). Using topic models with browsing history in hybrid collaborative filtering recommender system: Experiments with user ratings. International Journal of Information Management Data Insights, 1(2), 100027. ISSN 2667-0968.
12. Sengupta, S., Harris, M., Zhang, Y., & Owens, J. D. (2007). Scan primitives for GPU computing.
13. Galil, Z., & Italiano, G. F. (1991). Data structures and algorithms for disjoint set union problems. ACM Computing Surveys (CSUR), 23(3), 319-344.
14. Kushwaha, A. K., Kar, A. K., & Dwivedi, Y. K. (2021). Applications of big data in emerging management disciplines: A literature review using text mining. International Journal of Information Management Data Insights, 1(2), 100017. ISSN 2667-0968.
15. Goldschmidt, B., Schug, D., Lerche, C. W., Salomon, A., Gebhardt, P., Weissler, B., ... & Schulz, V. (2015). Software-based real-time acquisition and processing of PET detector raw data. IEEE Transactions on Biomedical Engineering, 63(2), 316-327.
16. Perez-Andrade, R., Cumplido, R., Feregrino-Uribe, C., & Del Campo, F. M. (2009). A versatile linear insertion sorter based on an FIFO scheme. Microelectronics Journal, 40(12), 1705-1713.
17. Ortiz, J., & Andrews, D. (2010, April). A configurable high-throughput linear sorter system. In 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW) (pp. 1-8). IEEE.

Copyright: *©2022 Anurag Dutta. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.*