International Journal of Innovative Research in Science and Engineering
Volume 7, Issue 03, 2021
www.ijirse.com
ISSN: 2454-9665

Algorithms as an Instrument of Information Technology

[1]Shagun Tyagi,
M. Tech, CSE Mewar University, Chittorgar, Rajasthan.
[2]Dr Ashish Chaturvedi,
Research Supervisor

1.1 Introduction

The word "algorithm" comes from the name of a Persian author, Abu Ja'far Mohammed ibn Musa al Khowarizmi (c. 825 A.D.), who wrote a textbook on mathematics. An algorithm is a way to accomplish any task in step by step process. Basically, algorithm is an approach to do any work efficiently and logically and at the same time it gives the sense to work without confusion and without being puzzled. Algorithms in fact are the ideas behind computer programs.

For an algorithm to be valid, each step (or instruction) must be:

- Unambiguous – the instruction can only be interpreted in one unique way
- Executable – the person or device executing the instruction must know how to accomplish the instruction without any extra information.
- Ordered – the steps of an algorithm must be ordered in a proper sequence to correctly accomplish the task.

1.2 Algorithms as a Technology

Computers are no doubt very fast, but they are not infinitely fast. Memory may be cheap, but not free. Computing time is in fact a bounded resource, and so is the space in memory. These resources should be used wisely, and algorithms that are efficient in terms of time or space will help us to do so.

1.2.1 Efficiency

In computer science, **algorithmic efficiency** is the property of an algorithm which relate to the amount of computational resources used by the algorithm. An algorithm must be analyzed to determine its resource usage. Algorithmic efficiency can be thought of as analogous to engineering productivity for a repeating or continuous process.

For maximum efficiency we wish to minimize resource usage. However, the various resources (e.g. time, space) cannot be compared directly, so which of two algorithms is considered to be more efficient often depends on which measure of efficiency is considered the most important, e.g. is the requirement for high speed, or for minimum memory usage, or for some other measure?

1.2.2 Measuring Time Requirements

How do you determine the time requirements of two algorithms?

- **Option 1: Write and Run a Program** - An obvious way to test the speed of two different algorithms would be to write two programs and then run these programs with some type of timing device. Although this can be a good indication of which algorithm is faster (we will often do this), there are at least four problems with this approach that need to be considered.

Therefore, our primary measurement of algorithm efficiency will be a mathematical analysis of the number of comparisons and/or assignments required by the algorithm to complete its task. In most cases, just comparisons will be used because a comparison (i.e. decision) is much more costly (in terms of time) than an assignment or calculation.
There are so many sorting algorithms. But in order to explain the importance of the efficiency of algorithms, we will see two algorithms for sorting. The first one is known as insertion sort, takes time roughly equal to cn^2 to sort n items, where c is a constant that does not depend on n. The second is merge sort which takes time roughly equal to *dnlgn*, where *lgn* stands for $log_2 n$ and d is another constant that also does not depend on n. Insertion sort usually has a smaller constant factor than merge sort, so that c < d.
For a concrete example, let us take a faster computer (computer A) running insertion sort against a slower computer (computer B) running merge sort. They each must sort an array of one million numbers. Suppose that computer A executes one billion instructions per second and computer B executes only ten million instructions per second, so that computer A is 100 times faster than computer B in raw computing power. To make the difference even more dramatic, suppose that the world's best programmer codes insertion sort in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers (Here c=2). Merge sort, on the other hand, is programmed for computer B by an average programmer using a high- level language with an inefficient compiler, with the resulting code taking *50nlgn* instructions (so that d=50). To sort one million numbers, computer A takes

$2*(10^6)^2$ instructions / 10^9 instructions per sec = 2000 seconds while computer B takes
$50.\ 10^6 \lg 10^6$ instructions /10^7instructions per sec. ≈ 100 seconds.

By using an algorithm whose running time grows more slowly, even with a poor compiler, computer B runs 20 times faster than computer A. Now we can see the importance of selection of good algorithm. The advantage of merge sort can easily be seen more when we sort ten million numbers: where insertion sort takes approximately 2.3 days, merge sort takes around 20 minutes. In general, as the problem size increases, we can easily understand the advantage of merge sort.

1.3 Analyzing Algorithms

Analyzing an algorithm means to predict the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware

ISSN: 2454-9665

are of primary concern, but most often it is computational time that we want to measure. Any algorithm normally is analyzed in terms of space and time. But we give the top priority to computational time which is the key factor to analyze an algorithm in a real sense.

Before we can analyze an algorithm, we must have a model of the implementation technology that will be used, including a model for the resources of that technology and their costs. Algorithms are the *only* important, durable, and original part of computer science *because* they can be studied in a machine independent and language independent way. We will do all our design and analysis for the RAM (random access machine) model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs.

- Each "simple" operation (+, -, =, if, call) takes exactly 1 step.
- Loops and subroutine calls are *not* simple operations, but depend upon the size of the data and the contents of a subroutine. We do not want ``sort" to be a single step operation.
- Each memory access takes exactly 1 step.

1.3.1 Theoretical analysis

In the theoretical analysis of algorithms, the normal practice is to estimate their complexity in the asymptotic sense, i.e. use Big O notation to represent the complexity of an algorithm as a function of the size of the input **n**. This is generally sufficiently accurate when **n** is large, but may be misleading for small values of **n** (e.g. bubble sort may be faster than quick sort when only a few items are to be sorted).

1.3.2 Measuring performance

For new versions of software or to provide comparisons with competitive systems, benchmarks are sometimes used, which assist with gauging an algorithms relative performance. If a new sort algorithm is produced for example it can be compared with its predecessors to ensure that at least it is efficient as before with known data—taking into consideration any functional improvements. Benchmarks can be used by customers when comparing various products from alternative suppliers to estimate which product will best suit their specific requirements in terms of functionality and performance. For example in the mainframe world certain proprietary sort products from independent software companies such as Syncsort compete with products from the major suppliers such as IBM for speed.

1.3.3 Implementation issues

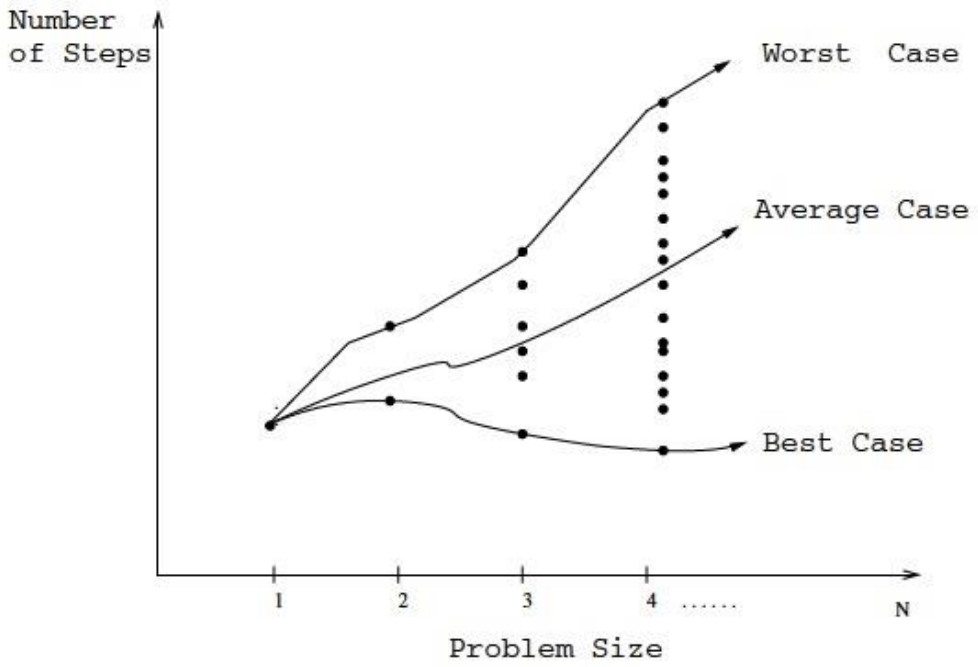
Implementation issues can also have an effect on actual efficiency, such as the choice of programming language, or the way in which the algorithm is actually coded, or the choice of a compiler for a particular language, or the compilation options used, or even the operating system being used. In some cases a language implemented by an interpreter may be much slower than a language implemented by a compiler.

ISSN: 2454-9665

There are other factors which may affect time or space issues, but which may be outside of a programmer's control; these include data alignment, data granularity, garbage collection, instruction-level parallelism, and subroutine calls.

1.4 Best, Worst, and Average Case

Using the RAM model of computation, we can count how many steps our algorithm will take on any given input instance by simply executing it on the given input. However, to really understand how good or bad an algorithm is, we must know how it works over all instances.
To understand the notions of the best, worst, and average-case complexity, one must think about running an algorithm on all possible instances of data that can be fed to it. For the problem of sorting, the set of possible input instances consists of all the possible arrangements of all the possible numbers of keys. We can represent every input instance as a point on a graph, where the *x*-axis is the size of the problem (for sorting, the number of items to sort) and the *y*-axis is the number of steps taken by the algorithm on this instance. Here we assume, quite reasonably, that it doesn't matter what the values of the keys are, just how many of them there are and how they are ordered. It should not take longer to sort 1,000 English names than it does to sort 1,000 French names, for example.



Best, worst, and average-case complexity

As shown in Figure 1, these points naturally align themselves into columns, because only integers represent possible input sizes. After all, it makes no sense to ask how long it takes to sort 10.57 items. Once we have these points, we can define three different functions over them:

ISSN: 2454-9665

- The *worst-case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size *n*. It represents the curve passing through the highest point of each column.
- The *best-case complexity* of the algorithm is the function defined by the minimum number of steps taken on any instance of size *n*. It represents the curve passing through the lowest point of each column.
- Finally, the *average-case complexity* of the algorithm is the function defined by the average number of steps taken on any instance of size *n*.

1.5 Classes of Algorithms

There are many ways to classify algorithms, and the merits of each classification have been the subject of ongoing debate. One way of classifying algorithms is by their design methodology or paradigm. There is a certain number of paradigms, each different from the other.

1.6 Sorting Algorithms

In computer science, sorting is an essential work for many applications towards searching and locating a prominent number of data. General description of sorting believed to be the process of rearranging the data into a particular order. The orders used are either in numerical order or lexicographical order. Sorting arranges the integer data into increasing or decreasing order and an array of strings into alphabetical order. It may also be called as ordering the data. Sorting is considered as one of the most fundamental tasks in many computer applications for the reason that searching a sorted array or list takes less time when compared to an unordered or unsorted list [68].

The main function of sorting algorithms is to place data elements of a list in a certain order. Sorting is one of the core computational algorithms used in many scientific and engineering applications. All sorting algorithms are problem specific means they work well on some specific problem and do not work well for all the problems. Some sorting algorithms are applied to small number of elements, some sorting algorithms are suitable for floating point numbers, some are fit for specific range, some sorting algorithms are used for large number of data, and some are used if the list has repeated values.

Moreover, each algorithm has its own advantages and disadvantages. For instance, bubble sort would be efficient to sort a small number of items. On the other hand, for a large number of items quick sort would perform very well. Therefore, it is not perpetually thinkable that one sorting method is better than another sorting method. Moreover the performance of each sorting algorithm relies upon the data being sorted and the machine used for sorting [69].

In general, simple sorting algorithms perform two operations such as compare two elements and assign one element. These operations proceed over and over until the data is sorted [70].

Moreover, selecting a good sorting algorithm depending upon several factors such as the size of the input data, available main memory, disk size, the extent to which the list is already sorted and the distribution of values [69].

We sort data either in numerical order or lexicographical, sorting numerical value either in increasing order or decreasing order and alphabetical value like addressee key. Many sequential sorting algorithms consume O (nlogn) time to sort n keys [1].

ISSN: 2454-9665

Sorting [3, 4] is defined as the operation of arranging an unordered collection of keys or elements into monotonically increasing (or decreasing) order. Specifically, S= {a_1, a_2 ……a_n} be a sequence of n elements in random order; sorting transforms S into monotonically increasing sequence S'= {a_1', a_2'…… a_n'} such that $a_i' \leq a_j'$ for $1 \leq i \leq j \leq n$, and S' is a permutation of S.

Sorting has two different meanings ordering and categorizing, ordering means to order the list of same items and categorizing means grouping and labeling the same type of items [2].

Sequential sorting algorithms are classified into two categories. The first category, "distribution sort", is based on distributing the unsorted data items to multiple intermediate structures which are then collected and stored into a single sorted list. The second one, "comparison sort", is based on comparing the data items to find the correct relative order [3].We focus on comparison based sorting algorithms. These algorithms use various approaches in sorting such as exchange, partition, and merge. The exchange approach repeats exchanging adjacent data items to produce the sorted list as in case of bubble sort [4]. The partitioning approach is a "divide and conquer" strategy based on dividing the unsorted list into two sub-lists according to a pivot elements selected from the list of keys. The two sub-lists are sorted and then combined producing the sorted list as in case of quick sort [5].

Merge sort approach is also a divide and conquer strategy that does not depend on a pivot element in partitioning process. The approach repeatedly divides the original list into sub-lists until the sub-lists have only one data item. Then these elements are merged together given the sorted list as in case of merge sort [6, 7]. Merge sort is frequently employed in many applications.

A sorting algorithm [12] is said to be in-place whenever it does sorting with the use of constant extra memory. In this case, the amount of extra memory required to implement the algorithm does not depend on the number of keys in the input list. In addition, the sorting algorithm is stable if it keeps the indices of a sequence of equal values in the input list (in any of the input list) in sorted order at the end of sorting. Otherwise, the algorithm is said to be unstable.

In designing parallel sorting algorithms, the fundamental issue is to collectively sort data owned by individual processors in such a way that it utilizes all processing units doing sorting work, while also minimizing the costs of redistribution of keys across processors. In parallel sorting algorithms there are two places where the input and the sorted sequences can reside. They may be stored on only one of the processor, or they may be distributed among the processors. Parallel merge sort on PRAM model was reported to have fast execution time of O (logN) for n input keys using N processors [8].

Instead of using multiprocessors to achieve parallelism, multi-core architectures are used to implement and executes parallelized applications. Several algorithms are introduced to solve this problem.

1.6.1 Sequential sorting algorithms

Sequential sorting algorithms are classified into two categories. The first category, "distribution sort", is based on distributing the unsorted data items to multiple intermediate structures which are then collected and stored into a single sorted list.

The second one, "comparison sort", is based on comparing the data items to find the correct relative order [22].

ISSN: 2454-9665

These algorithms use various approaches in sorting such as exchange, partition, and merge. The exchange approach repeats exchanging adjacent data items to produce the sorted list as incase of bubble sort [23].
The partitioning approach is a "divide and conquer" strategy based on dividing the unsorted list into two sub-lists according to a pivot element selected from the list. The two sub-lists are sorted and then combined giving the sorted list as in case of quick sort [24].
Merge approach is also a divide and conquer strategy that does not depend on a pivot element in portioning process. The approach repeatedly divides the original list into sub-lists until the sub-lists have only one data item. Then these elements are merged together given the sorted list as in case of merge sort [25, 26].

1.6.2 Parallelizing sorting algorithms

Parallelizing sorting algorithms needs a careful design to achieve well efficient results because of the high level date dependency evolved within these algorithms that exhibits parallelism. Sequential versions of bubble sort, quick sort and merge sort are parallelized using C ++ binding of MPI under MPICH2 for Windows. The "scatter/ merge" paradigm is used in parallelization.
The used paradigm has three fundamentals phases, scatter phase, sort phase and merge phase. The first phase is responsible for distributing the original unsorted data list among the MPI process in such a way each of them accepts a part of the original data to be manipulated with these parallel processes.
In sort phase, each process sorts its local unsorted data list using one of the selected sorting algorithms. All local sorted data are sent from these "slave" processes to only one process which serves as a master process to generate the sorted list in the merge phase.

1.7 Comparison of Sorting Algorithms

In the previous section, we have discussed the design and implementation of various algorithms considered for this study along with various test case experiments which are conducted for the evaluation of the practical efficiency of sorting algorithms. In this section we have discussed the experiment results besides analysis and comparison of various sorting algorithms with the help of experiment results. According to the theoretical study, we have concluded and compared the time complexity of various sorting algorithms in three different cases such as average, best and worst along with the stability and method of working. The table described below represents the variations, where n defines the number of items to be sorted, k defines the range of numbers in the list.

S. No.	Sorting Method	Best Case	Worst Case	Average Case	Stable	Method
1.	Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Yes	Exchange
2.	Modified Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	Yes	Exchange
3.	Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Yes	Selection
4.	Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	Yes	Insertion
5.	Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	Yes	Merge
6.	Quick Sort	O(nlogn)	$O(n^2)$	O(nlogn)	Yes	Partition

7.	Randomized Quick Sort	O(nlogn)	O(nlogn)	O(nlogn)	Yes	Partition
8.	Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	No	Selection

Table 1. Comparison of Comparison Based Sorting Algorithms

S. No.	Sorting Method	Best Case	Worst Case	Average Case	Stable
1.	Radix Sort	O(n*k/s)	O(n*k/s)	O(n*k/s)	No
2.	Counting Sort	$O(n+2^k)$	$O(n+2^k)$	$O(n+2^k)$	Yes
3.	Bucket Sort	O(n*k)	$O(n^2)$	O(n*k)	

Table 2. Comparison of Non-Comparison Based Sorting Algorithms

1.7.1 Problems and their Suitable Sorting Algorithm

Each sorting algorithm is well suited for certain problem this fact validates that sorting algorithms are problem specific. In the following table, according to [18] we have specified some problems and suggested the suitable sorting algorithms.

Problem Definition	Sorting Algorithms
Problems which do not require any extra memory to sort the array of data.	Insertion Sort, Selection sort, Bubble Sort.
Business applications and database applications which required a significant amount of extra memory to store data.	Merge Sort
Problems with the input is extracted from a uniform distribution of a range (0,1).	Bucket Sort
To sort the record with multiple fields and alphabet with constant size.	Radix Sort.
Problems with small input data sets.	Insertion Sort.
Problems with Large input data sets.	Merge Sort, Quick Sort, Heap Sort.
Problems with the repeated data items in the input array.	Counting Sort
To Sort the record based on address.	Bucket Sort.
Problems with the repeated data items in the input array and the resultant output should maintain the relative order of the data items with equal keys.	Bubble Sort, Merge Sort, Counting sort, Insertion Sort.
Problems with repeated data items in the input array and resultant output does not need to be maintained the relative order of the data items with equal keys.	Quick Sort, Heap Sort, Selection Sort.

ISSN: 2454-9665

Conclusion

The problem of sorting a set of objects is one of the first intensively studied computer science problems. Many of the best known applications of the Divide-and-Conquer design paradigm are sorting algorithms. During the 1960s, when commercial data processing became automated on a large scale, the sort program was the most frequently run program at many computer installations. The efficiency of data handling can often be substantially increased if the data are sorted according to some criteria of order. The final ordering of data can be obtained in a variety of ways, and only some of them can be considered meaningful and efficient.

There are several good reasons for studying sorting algorithms. First, they are of practical use because sorting is done often. Second, quite a lot of sorting algorithms have been devised and studying a number of them can take many different points of view towards the same problem. Third, sorting is one of a few problems for which we can easily derive strong lower bounds for worst case and average behavior.

There are two sorting methods:

- Internal sorting is used when the records fit in the computer's internal memory.
- Sorting that cannot be performed in main memory and must be done on disk or tape, is known as external sorting.

Here, we are more concerned with the internal sorting techniques.

REFERENCES

[1] MinsooJeon and Dongseung Kim, Load-Balanced Parallel Merge Sort on Distributed Memory Parallel Computers, IEEE, 2002.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 1990. ISBN 0-262- 03293-7. Chapter 27: Sorting Networks, pp.704–724S

[3] L. Rashid, W. Hassanein, M. Hammad "Analyzing and enhancing the parallel sort operation on multithreaded architectures" Journal of Supercomputing, vol. 53, no. 2, pp 293-312, 2010.

[4] O. Astrachan, "Bubble sort: An Archaeological Algorithmic Analysis" ACM SIGCSE Bulletin, vol. 35 no. 1, 2003.

[5] P. Tsigas and Yi. Zhang " A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on Sun Enterprise 10000" Proceedings of the 11th EUROMICRO Conference on Parallel Distributed and Network-Based Processing (PDP). pp. 372 – 381, 2003.

[6] A. LaMarca and R. E. Ladner "The influence of caches on the performance of sorting" Proceedings of 8th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA97), pp.370–379, 1997.

[7] B.R. Lyer and D.M. Dias "System Issues in Parallel Sorting for Database Systems" Proceedings of the International Conference on Data Engineering, pp. 246-255, 2003.

[8] R. Cole, "Parallel merge sort," SIAM Journal of Computing, vol. 17, no. 4, 1998, pp.770-785.

[9] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. Introduction to Algorithms (3rd ed.), MIT Press, 2009.

ISSN: 2454-9665

[10] Katajainen, Jyrki; Träff, Jesper L. A meticulous analysis of mergesort programs. Lecture Notes in Computer Science, 1997, Volume 1203/1997, 217-228.
[11] LaMarca, Anthony; Ladner, Richard. The influence of caches on the performance of sorting. Proc. 8th Ann. ACM-SIAM Symposium on Discrete Algorithms (SODA97), 370–379.
[12] D. E. Knuth, The Art of Computer Programming, Vol. IIISorting and Searching, Addison-Wesley, 1973.
[13] D. Geer. Chip Makers Turn to Multicore Processors. IEEE Computer, 38(5):11–13, 2005.
[14] G. Lowney. Why Intel is designing multi-core processors. https://conferences.umiacs.umd.edu/paa/lowney.pdf.
[15] J. Rattner. Multi-Core to the Masses. Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on, pages 3–3, 2005.
[16] L. Hammond, B. Nayfeh and K. Olukotun" A Single-Chip Multiprocessor" IEEE Computer, vol. 30 no. 9, pp 79-85, 1997.
[17] Intel Core2Duo.URL: http://www.intel.com/products/processor/core2duo/index.htm
[18] M. Kistler, M. Perrone, F. Petrini "Cell Multiprocessor Communication Network: