A Comprehensive Note on Complexity Issues in Sorting Algorithms

Parag Bhalchandra*, Nilesh Deshmukh, Sakharam Lokhande, Santosh Phulari*
*School of Computational Sciences, Swami Ramanand Teerth Marathwada University, Nanded, MS, India, 431606, Email: pub1976@ rediffmail.com

Abstract- Since the dawn of computing, the sorting problem has attracted a great deal of research. In past, many researchers have attempted to optimize it properly using empirical analysis. We have investigated the complexity values researchers have obtained and observed that there is scope for fine tuning in present context. Strong evidence to that effect is also presented. We aim to provide a useful and comprehensive note to researcher about how complexity aspects of sorting algorithms can be best analyzed. It is also intended current researchers to think about whether their own work might be improved by a suggestive fine tuning. Our work is based on the knowledge learned after literature review of experimentation, survey paper analysis being carried out for the performance improvements of sorting algorithms. Although written from the perspective of a theoretical computer scientist, it is intended to be of use to researchers from all fields who want to study sorting algorithms rigorously.
Keywords: Algorithm analysis, Sorting algorithm, Empirical Analysis Computational Complexity notations.

1. Introduction

Searching and Sorting are the tasks that are frequently encountered in various Computer Applications. Since they reflect fundamental tasks that must be tackled quite frequently, researchers have attempted in past to develop algorithms efficient in terms of optimum memory requirement and minimum time requirement i.e., Time or Space Complexities. Together with searching, sorting is probably the most used algorithm in Computing, and one in which, statistically, computers spend around half of the time performing [a]. Sorting algorithms are always attractive because of the amount of time computers spend on the process of sorting has always been a matter of research attention. For this reason, the development of fast, efficient and inexpensive algorithms for sorting and ordering lists and arrays is a fundamental field of computing. By optimizing sorting, computing as a whole will be faster. When we look to develop or use a sorting algorithm on large problems, we came across previous research literature where it was mentioned clearly to concentrate on how long the algorithm might take to run. We discovered that, the time for most sorting algorithms depends on the amount of data or size of the problem and in order to analyze an algorithm, we required to find a relationship showing how the time needed for the algorithm depends on the amount of data. We found that, for an algorithm, when we double the amount of data, the time needed is also doubled. The analysis of another algorithm told us that when we double the amount of data, the time is increased by a factor of four. The latter algorithm would have the time needed increase much more rapidly than the first. We have discovered that, some factors other than the sorting algorithm selected to solve a problem, affect the time needed for run [1]. It is just because different people carrying out a solution to a problem may work at different speeds, even when they use the same sorting method, as different computers work at different speeds. The different speeds of computers can be due to different "clock speeds", the rates at which steps in the program are executed by the machine and different "architectures," the way in which the internal instructions and circuitry of the computer are organized. Consequently, analysis of sorting algorithm can not predict exactly how long it will take on a particular computer. We also found that, the analysis of efficiency depends considerably on the nature of the data. For example, if the original data set is almost ordered already, a sorting algorithm may behave rather differently than if the data set originally contains random data or is ordered in the reverse direction. The purpose of this investigation is to magnify analysis of sorting algorithms considering all possible factors and make a concise note of it. Our work may be useful for some applications that seek to determine which sorting algorithm is the fastest to sort the lists of different lengths, and, to, therefore determine which algorithm should be used depending on the list length. For example Shell sort should be used for sorting of small (less than 1000 items) arrays. It has the advantage of being an in-place and non-recursive algorithm, and is faster than Quicksort up to a certain point. For larger arrays, the best choice is Quicksort, which uses recursion to sort lists, leading to higher system usage but significantly faster results. We have attempted to review the rich body of sorting literature in accord with their utility and performance so as to make a critical analysis of them in order to discover tuning factors. These factors are intended to help the reader to avoid wasted efforts in order to produce correct complexity values. Most of the part of this paper concentrates on the study of algorithms for problems in the standard format where both instances and outputs are finite objects, and the key metrics are resource usage (typically time and memory).Several of the suggestions enunciated here may be somewhat controversial, but we have, at least elaborated them. Indeed, although there is much common agreement on what makes good experimental analysis of sorting algorithms, certain aspects have been the subject of debate, such as the relevance of running time comparisons.

2. Background Knowledge

In computer science and mathematics, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order. Sorting algorithms are often prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts. Herein, we restrict the scope of sorting to ordering of data by a digital computer. Given a collection of data entries and an ordering key, sorting deals with various processes invoked to arrange the entries into a desired order. Sorting algorithms are of two types. Internal and External, depending upon ordering a list of elements residing in primary storage or secondary storages. There are two types of each of them viz, the comparative and the distributive. The comparative algorithms order the list by making a series of comparisons of the relative magnitude of the ordering keys of the elements. The distributive algorithms order the list by testing a key or a digit of a key against a standard and collecting all members of a group together. Group definitions are then modified so that all elements and groups are ordered during a last pass. The performance of comparative algorithms varies with the number of elements to be sorted and the permutation of the elements. The performance of distributive algorithm varies with the range of the keys and their distribution. The criteria for measuring the performance of an ordering algorithm include, the number of comparisons that must be performed before the list is ordered, the number of movements of data on the list before the list is ordered, the amount of space required beyond that needed to hold the list, and the sensitivity to certain kinds of order of the data. The number of comparisons among algorithms varies considerably. A minimum storage algorithm is one that requires little or no additional storage to perform the ordering. Algorithmic complexity of sorting algorithm is generally written in a form known as Big-O notation, where the O represents the complexity of the algorithm and a value n represents the size of the set the algorithm is run against. For example, $O(n)$ means that an algorithm has a linear complexity [2]. Generally, the complexity notational terminology is covered as in [3]. Research on efficiency analysis of sorting algorithms [4] uses Big Oh (O), Omega (Ω) and Theta (Θ) notations to give asymptotic upper, lower, and tight bounds on time and space complexity of sorting algorithms. The best and worst cases in a given algorithm express what the resource usage is *at least* and *at most*, respectively. An algorithm's average performance is its behavior under "normal conditions". In almost all situations the resource being considered is running time, but it could also be memory, for instance. The worst case is most of concern since it is of critical importance to know how much time would be needed to guarantee the algorithm would finish. Let us see how complexity of a sorting algorithm is measured [1] .Consider Merge Sort algorithm. The merge sort function/algorithm (merge sort l) takes a list l of length n, and does a merge on the merge sort of the first half of l, and the merge sort of the second half of l. The stopping condition is when the list l is of size 0 or 1. Let the merge function takes two sorted lists l1 and l2. At each step merge takes the smaller of the head of l1 and the head of l2, and appends it to a growing list, and removes that element from the list (either l1 or l2). A merge on lists of length n/2 is O (n).The running time of merge sort on a list of n elements is then , $t(0) = 0$, $t(1) = 1$, …… $t(n) = 2.t(n/2) + c.n$, where c.n is the cost of merging two lists of length n/2, and the term 2t(n/2) is the two recursive calls to merge sort with lists l1 and l2 of length n/2. Consequently,

$$\begin{aligned} T(n) &= 2.t(n/2) + c.n \\ &= 2.(2.t(n/4) + c.n/2) + c.n \\ &= 2.(2.(2.t(n/8) + c.n/4) + c.n/2) + c.n \\ &= 8.t(n/8) + 3.c.n \end{aligned}$$

A pattern emerges and by induction on i we obtain

$t(n) = 2^i.t(n/2^i) + i.c.n$, Where the operator ^ is "raised to the power".

If we assume that n is a power of 2 (i.e., 2, 4, 8, 16, 32, generally 2^k) the expansion process comes to an end when we get t (1) on the right, and that occurs when i=k, whereupon

$t(n) = 2^k.t(1) + k.c.n$

We have just stated that the process comes to an end when i=k, where n = 2^k. Put another way, k = log n (to the base 2 of course), therefore

$t(n) = n + c.n.\log n = O(n \log n)$.

Thus O(n log n) is the threshold value of complexity of sorting algorithms.

In our work, if the size of unsorted list is (n), then for typical sorting algorithm, good behavior is O (n log n) and bad behavior is $\Omega(n^2)$. The Ideal behavior is O (n). Sort algorithms which only use an abstract key comparison operation always need Ω ($n \log n$) comparisons in the worst case. Literature review carried out in [5] indicates the man's longing efforts to improve running time of sorting algorithm with respect to above core algorithmic concepts.

In addition to algorithmic complexity, the speed of the various sorts can be compared with empirical data. Since the speed of a sort can vary greatly depending on what data set it sorts, accurate empirical results require several runs of the sort be made and the results averaged together. We feel that this description is slightly inaccurate, since the running time can significantly deviate from a precise proportionality, especially for small *n*. Technically, it's only necessary that for large enough *n*, the algorithm takes more than *an* time and less than *bn* time for some positive real constants *a , b*. Keeping in mind this discussion on the current practices to analyze sorting algorithms , we can say that for a given sorting algorithm, it can be proven that there exists an order of number which this sorting algorithm will execute in linear time. However, for a general case, we agree that, no sorting algorithm can perform better than n (log n) [6]. At last but not the least , we take an opportunity to quote that , even though Linear time is often viewed as a desirable attribute for a sorting algorithm ,much research has been invested into creating algorithms exhibiting (nearly) linear time or better. These researches included both software and hardware methods. In the case of hardware, some algorithms which, mathematically speaking, can never achieve linear time with the standard computation model are now able to run in linear time. We found that there are several hardware technologies which exploit parallelism to provide this. An example is associate memory. [17]

3. Some Light on Proper Tuning of Sorting Algorithm's Analysis

In above discussions, by analysis of algorithm we meant theoretical and algorithmic analysis only. Generally empirical analysis of sorting algorithms is considered to be easy, but that it is in fact difficult and requires a place in research topics. Empirical analysis of algorithms is also an important idea in its own right. Theoretical analysis does not give much of an idea of how well a given algorithm will perform in a specific situation [7], empirical analysis would help here. Empirical analysis is also important in comparing two algorithms which may or may not have the same order of complexity – when would one use one and not the other. As an example of this consider Insertion Sort and Quick sort, where for small input instances Insert Sort could be a better algorithm to use and is certainly easier to understand and code (and has a good best case performance). We believe that the improvement of efficiency of a sorting algorithm is a continuing process. A large body of literature about sorting has been developed as the result of continuous and intensive work in the area since the invention of the general-purpose digital computer. At different times during the history of sorting, workers in the field were preoccupied with different problems. In the late 1950's, concern was with improved techniques using tape drives; in the early 1960's, with efficient methods using minimum storage space; in the mid 1960's, with disk-oriented methods; and currently the industry is becoming concerned with sorting on parallel processors and in virtual memory environments. Work in sorting is progressing along several lines. Some effort is aimed at developing greater insight into known techniques and at discovering more details about their behavior in different situations. A second line is the development of improved techniques. For example, the search for algorithms combining efficient use of storage space with a small number of comparisons has resulted in significantly different techniques from those that appear as "standard" in the early literature. Other activity is concerned not so much with the fundamental techniques of achieving order but with the environment in which an ordering process occurs. Investigations of new kinds of devices, new data-handling techniques for new devices, new processor or channel architectures, etc., are constantly underway. A very thorough understanding of sorting is based on a usable knowledge of these disciplines. But the development of sorting programs is an activity far more extensive than the development of sorting algorithms. The worker with little mathematics or statistics can make important contributions to the field once he has understood an algorithm theoretically developed. Therefore, those who wish only to find and implement reasonable sort knowledge need no associated specialized knowledge. Working descriptions of sorting methods with usable guides to relative performance exist in the extensive literature of the field. We feel that, the contributions made by non mathematicians may be mere superficial. Though, many of the articles tend to be oriented toward statisticians or mathematicians, there exists sufficient narrative material so that a programmer or analyst without this background can familiarize himself with techniques and alternatives. Thus we are indulged in a dilemma of authentic work. We feel that, those who desire to specialize in the development or analysis of sort algorithms or to be very careful in their choice of a sort procedure must have a statistical and mathematical background. An appreciation of the derivation, applicability, and generality of formulas used to project performance requires concepts of permutation, distribution, randomness, non parametric tests for randomness, autocorrelation, etc. Developing performance analysis methods for new techniques or new combinations of techniques requires facility in algebra and calculus. Algebra is often used to describe sorting processes. Bounds or limits on performance are often expressed in the calculus. The past emphasis among theoreticians on the more rigorous and theoretical modes of analysis is of course to be expected. Moreover, it has strong justifications. To this day it is difficult to draw useful extrapolations from experimental studies. Indeed, it was the lack of scientific rigor in early experimental work that led Knuth and other researchers in the 1960's to emphasize worst- and average-case analysis and the more general conclusions they could provide, especially with respect to asymptotic behavior. The benefits of this more mathematical approach have been many, not only in added understanding of old algorithms but in the invention of new algorithms and data structures, ones that have served us well as faster machines and larger memories have allowed us to attack larger problem instances. Almost all theoretical computer scientists would choose experimental analysis, but such gets often treated almost as an afterthought. Indeed, experimental analysis of sorting algorithms has been almost common these days to distinguish between different approaches to analyze invisible in the theoretical computer science literature. We saw experimental work dominates algorithmic research in most other areas of computer science and related fields such as operations research. Recently, however, there has been an upswing in interest in experimental work in the theoretical computer science community. This is not because theorists have suddenly discovered a love of programming or because they are necessarily good at it, but because of a growing recognition that theoretical results cannot tell the full story about real-world algorithmic performance. Encouragement for experimentation has come both from those like ourselves who are trying to experimenting and from funding agencies who view experimentation as providing a pathway from theory into practice. We are interested in influences that go in the other direction. How can theoretician's concern for asymptotic, generality, and understanding can help to derive a more scientific approach and how can expertise be obtained in the form of fine tuning from the background in theoretical analysis. It is this scientific bias that we will be stressing in this part, while also providing more general advice for would be can such expertise helps in doing experimentation? Unfortunately, as many researchers have already discovered, the field of experimental analysis of sorting algorithm is fraught with pitfalls. In many ways, the implementation of a sorting algorithm is the easy part. The hard part is successfully using that implementation to produce meaningful and valuable (and publishable!) research results. Although much of what we have to say here is due to the opportunity to read in the experimental literature, with special emphasis on papers about the Sorting Problem, which we have surveyed as annexed here. Literature review makes impression that the sorting algorithms have widely analyzed using criterion discovered by Baase [8], which are Correctness, Work done, Space used, Simplicity or clarity and Optimality. Similarly Sedgewick [9] devotes a chapter to the "Implementation of Algorithms" Here he makes the claim that "it is unfortunately all too often the case that mathematical analysis can shed very little light on how well a given algorithm can be expected to perform in a given situation". He stresses the importance of empirical analysis in this case. He also advocates the use of empirical analysis in comparing two algorithms to solve the same problem – "run both to see which takes longer". Moret and Shapiro [10] do not just present algorithms in pseudocode but give actual running programs in Pascal. They give three reasons for doing this. Two of their reasons have relevance here – that the distance from the pseudocode description of an algorithm to its implementation is often considerable and requires nontrivial decisions; and that where algorithms with similar asymptotic behaviors have been proposed for the same problem then an informed choice can only be made by implementing and comparing them. Chapter 8 of their book "Sorting: A case study in efficient coding" gives some insight into the complexity of the task of empirical analysis. Brunskill and Turner [11] give a list of some things that the execution time of a given program will depend on the CPU ,the compiler , the programming language ,the way the program is constructed , time for disk accesses and other IO , whether the system is single or multitasking. They do not discuss in any detail how these things would / could affect the program or what is required to understand them. A combined reading of Baase, Sedgewick, Moret & Shapiro and Brunskill & Turner create an impression that the analysis of sorting algorithms carried out previously mere on theoretical basis or experimentally emphasizing running time, suffers a fine tuning. In order to speak correctly about the complexity aspects, we need to understand a number of areas and the interplay between them clearly. This gives some insight into the difficulty of the problem of correctly formulating complexity values. This is the reason why we consider empirical analysis of sorting algorithms a crucial part of the analysis of algorithms. Even many course curriculums on analysis of algorithms expects that topics like Divide and Conquer, Amortized Analysis, Greedy Approaches need to be taught through the perspective of analysis of Sorting Algorithms. We are of the opinion that beside above discussions a contribution from Sedgewick and Flajolet [12] is most important for proper tuning. In their approach, Sedgewick and Flajolet made it clear that doing empirical analysis of sorting algorithms properly is a non-trivial exercise.

In order to do proper analysis we need to

a. understand the theoretical analysis
b. decide on what should be measured
c. decide on appropriate hardware
d. decide on an appropriate implementation language
e. decide on appropriate data structures
f. implement the algorithms
g. implement some form of timing device
h. Create the input data sets necessary to produce the measurements we need

i. measure the performance of the algorithm on the different input data sets
j. interpret the results and relate the results to the theoretical analysis

Few of these tasks are trivial. To deal with them adequately, knowledge and understanding of a number of theoretical concepts / areas is required including asymptotic notation, probability theory, machine architecture, programming languages, compilers, data structures and machine representation of these, and experimental statistics. In addition, programming and presentation skills are necessary to complete the task successfully. We admire that, it is essential to be aware of them and to be able to determine which require more attention in a particular case. Previous researchers did not consider all of these aspects in every empirical analysis and hence those results are rather relative than absolute. To support our opinion we consider an example based on the probability of the number being searched for being in each position in the list or not in the list at all. Clearly an understanding of asymptotic notation and some probability theory is required to understand this analysis. This implies that, in our sorting experiments, we must decide that, for each instance size we need to test every possibility and then average the resulting running times for each case. This would be the most exact thing to do but it would probably not be practical for any reasonably sized lists. We, thus, have to try to approximate this average case performance. This, hence, means making decisions- which must be clear and well motivated – about how many cases to test and how to generate these test cases that too having an understanding of probability theory and statistics as well as perhaps knowing about pseudo-random numbers generators is essential. In addition, some understanding of the machine architecture is required to make sure that we can handle the test cases we decide on. Once we have made these decisions we still have lot of other factors to consider. Since past researchers of sorting theory have not looked upon this sphere of intelligence, we doubt in the absolute values. Further , even though the effective way to compare how different algorithms perform on a system, the main disadvantage of empirical data we have observed , that it is entirely dependent on the computer where it has been obtained on. Very different results can arise from running algorithms on systems as dissimilar as a mainframe and a cell phone. Different variables, such as memory, processor, operating system, and currently running programs can affect the runtime of the algorithms. Even though they are kept constant in an investigation, by always running all the algorithms on the same system, nonetheless, we will not come to any system-independent conclusions. Many research scholars have restricted their work to Algorithmic best, worst and average cases. We found that, the average and worst-case performances are mostly used in sorting algorithm analysis while best-case performance is more of a fantasy description of a sorting algorithm. Computer scientists use probabilistic analysis techniques, especially expected value, to determine expected average running times. Similarly, worst case performance analysis is often easier to do than "average case" performance. Determining what "average input" is a bit difficult, and often that "average input" has properties which make it difficult to characterize mathematically. Similarly, even when a sensible description of a particular "average case", which will probably only be applicable for some uses of the algorithm, is possible, they tend to result in more difficult to analyze equations. For many sorting algorithms, it is difficult to analyze the average-case complexity. Generally speaking, the difficulty comes from the fact that one has to analyze the time complexity for all inputs of a given length and then compute the average. This is in fact a difficult task. Researchers have tackled it by using the incompressibility method, where we can choose just one input as a representative input and via Kolmogorov complexity, we can show that the time complexity of this input is in fact the average-case complexity of all inputs of this length [13]. However in our opinion, constructing such a "representative input" is impossible, but many times we know it exists and this is sufficient for the proper analysis. The price of this generality is exponential complexity; with the result that many problems of practical interest are solvable better than mare such knowledge of sorting. For these reasons, the analysis for sorting algorithms often considers separate cases, depending on the original nature of the data or the method deployed with optional considerations of the under laying hardware. We observed that the limitations of computational capacity prevent them from being solved in practice. The increasing diversity in computing platforms motivates consideration of multi-processor environment. Literature review suggests that no substantial efforts were found mentioned regarding complexity in multiprocessor environment in past time .Recently, many results on the computational complexity of sorting algorithms were obtained using Kolmogorov complexity (the incompressibility method). Especially, the usually hard average-case analysis is amenable to this method. A survey [13] shows such results about Bubble sort, Heap sort, Shell sort, with stacks and queues in sequential or parallel mode. It is also found that the trade-off between memory and sorting is enhanced by the increase in availability of computer memory and the increase in processor speed. Currently, the prices of computer memory are decreasing. Therefore, acquiring larger memory configurations is no longer an obstacle, making it easier to equip a computer with more memory. Similarly, every year there is an increase in computer speed. Increasing computer speed causes acceleration of comparison based algorithms. Thus knowledge proved by some researcher for a sorting algorithm on its complexity can not be considered absolute as it may be increased or decreased analogously. We are also of the opinion that, the programming language and compiler / interpreter used can also affect the speed of programs. A research [14] used Python, but if the algorithms were implemented in a language such as C, the result would have probably been much faster sorting, due to C's compiled nature over Python's interpreted approach. Likewise, if the language the sorting algorithm is to be implemented with is particularly efficient for some aspects like recursion, Quick sort might have gotten a much smaller runtime. This leads to another limitation of sorting investigation as past researchers did not conclude anything about code-independent algorithms. In our view, the solution to this drawback would be to use a system-independent method of analyzing algorithms by using asymptotic analysis. By obtaining relevant data from the analysis of algorithms, a concrete comparison regarding their speed can be used to obtain system and programming language independent results. Thus previous research has limitation of a runtime-based comparison. Similarly, we found other limitation of a comparison based on random arrays. Major sorting algorithms investigated so far are on purely random arrays. Such analysis may show the expected time needed for an algorithm to sort a random list. However, most of the time lists in computing are not entirely random. Usually, long lists are not created from scratch, but rather they are continuously being created by adding items to it. Therefore, most unsorted lists in computing will actually be mostly sorted. This kind of list, however, was not explicitly studied in any investigation. Apart from only random and mostly sorted lists, arrays sorted in reverse order and arrays with duplicate elements could also have been investigated to lead to more thorough conclusions on the best algorithm.

4. Analysis of Sorting Algorithms

In the context of Reviewed Literature and Discovery of Suggestive Tuning Factors

Keeping in mind discussions of introductory part, we now speak on complexity issues in sorting algorithms. Although asymptotic analysis of the algorithms is touched upon herein, the main type of comparison discussed is an empirical assessment based on running each algorithm against random lists of different sizes. We have borrowed some readymade results of known and authentic work [14,15,16] so as to avoid reparative findings as nothing is said beyond O(n log n) as far as sorting algorithms are concerned [6] .This reduces the length of the paper there by making a concise representation. For simplicity, we assume that the common sorting algorithms can be divided into two classes by the complexity of their algorithms as,

(n^2), which includes the bubble, insertion, selection, and shell sorts , and

(n log n) which includes the heap, merge, and quick sorts.

A) Bubble Sort

The bubble sort is the oldest and simplest sort in use. Unfortunately, it's the slowest one. The bubble sort works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order). This causes larger values to "bubble" to the end of the list while smaller values "sink" towards the beginning of the list. The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage. While the insertion, selection and shell sorts also have O (n^2) complexities, they are significantly more efficient than the bubble sort. A fair number of algorithm purists (which means they've probably never written software for a living) claim that the bubble sort should never be used for any reason. Realistically, there isn't a noticeable performance difference between the various sorts for 100 items or less, and the simplicity of the bubble sort makes it attractive. The bubble sort shouldn't be used for repetitive sorts or sorts of more than a couple hundred items. Clearly, bubble sort does not require extra memory.

B) Selection Sort

The selection sort works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled. The selection sort has a complexity of O (n^2). It is simple and easy to implement and it is Inefficient for large lists, so similar to the more efficient insertion sort that the insertion sort should be used in its place. The selection sort is the unwanted stepchild of the n^2 sorts. It yields a 60% performance improvement over the bubble sort, but the insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort. In short, there really isn't any reason to use the selection sort - use the insertion sort instead. The worst case occurs if the array is already sorted in descending order. The Selection sort spends most of its time trying to find the minimum element in the "unsorted" part of the array. It clearly shows the similarity between Selection sort and Bubble sort. Bubble sort "selects" the maximum remaining elements at each stage, but wastes some effort imparting some order to "unsorted" part of the array. Selection sort is quadratic in both the worst and the average case, and requires no extra memory. We highlight that these observations hold no matter what the input data is. In the worst case, this could be quadratic, but in the average case, this quantity is O (*n* log *n*). It implies that the running time of Selection sort is quite insensitive to the input. If you really want to use the selection sort for some reason, try to avoid sorting lists of more than a 1000 items with it or repetitively sorting lists of more than a couple hundred items.

C) Insertion Sort

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place. Like the bubble sort, the insertion sort has a complexity of $O(n^2)$. Although it has the same complexity, the insertion sort is a little over twice as efficient as the bubble sort. It is relatively simple and easy to implement and inefficient for large lists. Best case is seen if array is already sorted. It is a linear function of *n*. The worst-case occurs; when array starts out in reverse order .It is a quadratic function of *n*. The insertion sort is a good middle-of-the-road choice for sorting lists of a few thousand items or less. The algorithm is significantly simpler than the shell sort, with only a small trade-off in efficiency. At the same time, the insertion sort is over twice as fast as the bubble sort and almost 40% faster than the selection sort. The insertion sort shouldn't be used for sorting lists larger than a couple thousand items or repetitive sorting of lists larger than a couple hundred items. Since multiple keys with the same value are placed in the sorted array in the same order that they appear in the input array, Insertion sort is stable. This algorithm does not require extra memory.

D) Shell Sort

Invented by Donald Shell in 1959, the shell sort is the most efficient of the $O(n^2)$ class of sorting algorithms. It is a "diminishing increment sort", better known as a "comb sort" to the unwashed programming masses. The algorithm makes multiple passes through the list, and each time sorts a number of equally sized sets using the insertion sort. The size of the set to be sorted gets larger with each pass through the list, until the set consists of the entire list. (Note that as the size of the set increases, the number of sets to be sorted decreases.) This sets the insertion sort up for an almost-best case, run each iteration with a complexity that approaches $O(n)$. It is efficient for medium-size lists. It is somewhat complex algorithm, not nearly as efficient as the merge, heap, and quick sorts. The function form of the running time for all Shell sort depends on the increment sequence and is unknown. For the above algorithm, two conjectures are $n(\log n)^2$ and $n^{1.25}$. Furthermore, the running time is not sensitive to the initial ordering of the given sequence, unlike Insertion sort. The shell sort is by far the fastest of the N^2 class of sorting algorithms. It's more than 5 times faster than the bubble sort and a little over twice as fast as the insertion sort, its closest competitor. The shell sort is still significantly slower than the merge, heap, and quick sorts, but its relatively simple algorithm makes it a good choice for sorting lists of less than 5000 items unless speed is hyper-critical. It's also an excellent choice for repetitive sorting of smaller lists.

E) Quick Sort

The quick sort is an in-place, divide-and-conquer, massively recursive sort. As a normal person would say, it's essentially a faster in-place version of the merge sort. The quick sort algorithm is simple in theory, but very difficult to put into code .This recursive algorithm consists of making decisions based on the pivot element. It then splits the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot. If there are one or less elements in the array to be sorted, then returns immediately. The efficiency of the algorithm is majorly impacted by which element is chosen as the pivot point. The worst-case efficiency of the quick sort, $O(n^2)$, occurs when the list is sorted and the left-most element is chosen. Randomly choosing a pivot point rather than using the left-most element is recommended if the data to be sorted isn't random. As long as the pivot point is chosen randomly, the quick sort has an algorithmic complexity of $O(n \log n)$. It is extremely fast. It is very complex algorithm, massively recursive. The running time of quick sort depends on whether partition is balanced or unbalanced, which in turn depends on which elements of an array to be sorted are used for partitioning. A very good partition splits an array up into two equal sized arrays. A bad partition, on other hand, splits an array up into two arrays of very different sizes. The worst partition puts only one element in one array and all other elements in the other array. If the partitioning is balanced, the Quick sort runs asymptotically as fast as merge sort. On the other hand, if partitioning is unbalanced, the Quick sort runs asymptotically as slow as insertion sort. The best thing that could happen in Quick sort would be that each partitioning stage divides the array exactly in half. The quick sort is by far the fastest of the common sorting algorithms. It's possible to write a special-purpose sorting algorithm that can beat the quick sort for some data sets, but for general-case sorting there isn't anything faster. We conclude with an observation. It has been brought to our notice that the empirical data obtained reveals the speed of each algorithm, from slowest to fastest , for a sufficiently large list, ranks as 1) Quicksort , 2)Shell sort , 3) Selection sort , 4)Insertion sort , 5)Bubble sort. There is a large difference in the time taken to sort very large lists between the fastest two and the slowest three. This is due to the efficiency Quicksort and Shell sort have over the others when the list sorted is sufficiently large. Also, the results show that for a very small list size, only selection sort and insertion sort are faster than Quick and Shell sort, and by a very small amount. It should also be noted that Quicksort is eventually faster than Shell sort, though it is slower for small lists. In a practical sense, the difference between the speeds of Quicksort and Shell sort are not noticeable unless the list is very large (has over 1000 items). For very small lists, the difference between all the algorithms is too small to be noticeable. However, Shell sort is much less system-intensive than Quicksort because it is not recursive. Considering all this, for lists expected to be less than 1000 items, Shell sort is the optimal algorithm. It is in- place, non-recursive, and fast, making it sufficiently powerful for everyday computing. For lists expected to be very large (for example, the articles in a news website's archive, or the names in a phonebook) Quicksort should be used because, despite its larger use of space resources, it is significantly faster than any other of the algorithms in this investigation when the array is sufficiently large

5. Epilogue

Past researches on sorting algorithms have more emphasis on theoretical and empirical analyses. We regret that the algorithmic study based mere on time of execution of sorting random lists alone is not complete. Although speed of algorithms is a very important factor, it is not the only factor that must be taken into account when comparing sorting algorithms and saying that “X” is faster than “Y” and hence recommended for a “Z” situation. There are many other aspects that need to be taken into account, such as memory usage, CPU usage, algorithm correctness, code reusability, et cetera. Though sorting algorithms are blazingly fast, that speed comes at the cost of complexity. Recursion, advanced data structures, multiple arrays, etc make extensive use of those nasty things. Keeping note of these points, we have identified some intelligent tuning factors. Through our extended paper, we have come with a new perspective to do a comparison of these factors so as to determine which algorithm, as a whole, is most efficient. But as with everything else in the real world, there are trades-offs so do with the sorting algorithms. Thus knowledge proved by some researcher for a sorting algorithm on its complexity can not be considered absolute as it may be increased or decreased analogously.

Conclusion

In our paper, asymptotic analysis of the algorithms is mainly touched upon and efforts are made to point out some deficiencies in earlier work related to analysis of sorting algorithms. Till today, sorting algorithms are open problems and in our view, complexity research regarding sorting algorithm, up to some extent, is the momentarily belief among people. These researches are not absolute as their results are specific some factors discussed herein. We have shown that, every sorting algorithm can undergo a fine tuning with the intelligence aspects we have discovered so as to gain significant reduction in complexity values. The important thing we want to share is to forget the prejudice i.e., pick the sorting algorithm that we think is most appropriate for the task at hand, there by neglecting its literature values as those values are not absolute, rather relative. We are aware that the efficiency gain will not go beyond O (n log n), but hopeful enough to reduce complexities by using intelligent tactics , for example, there could be a smooth transition from quadratic complexity to linear one observed in comparative sorts due to intelligently using linked lists instead of arrays to hold data . This drastically reduce the space requirement since no need to swap the data as we need to change the pointers only , there by keeping the contents of nodes , the same . We have also showed that the choice of sorting algorithm is not a straight forward matter, as a number of issues may be relevant. It may be the case that an O (n*n) algorithm is more suitable than an O (n log n) algorithm. Some factors may be the quality of object code, computing platforms available, size of objects to be swapped, number of times algorithm is to be used versus time to develop (if not already in place), criticality of run time (maybe we don't care), size of input.

References

[1] Joyannes Aguilar, 2003, 360

[2] Darlington J. (1978) Acta Inf. II, 1-30.

[3] Andersson T., Nilsson H.S. and Raman R. (1995) Proceedings of the 27th Annual ACM Symposium on the Theory of Computing.

[4] Liu C. L. (1971) Proceedings of Switching and Automata Theory, 12th Annual Symposium, East Lansing, MI, USA , 207-215.

[5] Sedgewick (1997) Talk presented at the Workshop on the probabilistic analysis of algorithms, Princeton.

[6] Nilsson S. (2000) Doctor Dobbs Journal.

[7] Richard Harter. (2008) ERIC Journal Number 795978, Computers & Education, v51 n2, 708-723.

[8] Baase S., Computer Algorithms*:* Introduction to Analysis and Design, Addison Wesley, Reading, Massachusetts, second edition, 1988.

[9] Sedgewick R., Algorithms, Addison-Wesley, Reading, MA, second edition, 1988.

[10] Moret B.M. E., Shapiro H. D., Algorithms from P to NP: Volume I, Design and Efficiency, Benjamin Cummings, Redwood City, CA, 1991.

[11] Brunskill D. Turner J. (1997) Understanding Algorithms and Data Structures, MCGraw-Hill, Maidenhead, England.
[12] Sedgewick R., Flajolet P. An Introduction to the Analysis of Algorithms, Addison-Wesley, Reading, MA, 1996.
[13] Paul Vitanyi (2007) Analysis of Sorting Algorithms by Kolmogorov Complexity (A Survey), appeared in Entropy, Search, Complexity, Bolyai Society Mathematical Studies, Eds., Springer-Verlag, 209—232.
[14] Juliana Pena Ocampo, An empirical comparison of the runtime of five sorting algorithms, International Baccalaureate Extended Essay, Clegio Colombo Britanico, Santiago DeCali, Colombai, English version 2008
[15] Parag Bhalchandra, Proliferation of Analysis of Algorithms with application to Sorting Algorithms, M.Phil Dissertation, VMRF, India, July 2009
[16] John Harkins, Tarek El-Ghazawi, Esam El-Araby, Miaoqing Huang, Performance and Analysis of Sorting Algorithms on the SRC 6 Reconfigurable Computer, The George Washington University , USA
[17] Li Xiao, Xuedong Zhang, Stefan A. (2000) ACM Journal on Experimental Algorithmics, 5 (3), 1-22.