ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 2, Issue 3, pp: (394-398), Month: July - September 2014, Available at: www.researchpublish.com

Performance Comparison of Sorting Algorithms On The Basis Of Complexity

[1]Mr. Niraj Kumar, [2]Mr. Rajesh Singh

[1, 2](Assistant Professor) Department of Computer Science & Engineering
R.T.C Institute of Technology, Ranchi, India

Abstract: **When using the visualize to compare algorithms, never forget that the visualize sorts only very small arrays. The effect of quadratic complexity (either a square number of moves or a square number of exchanges) is dramatic as the size of the array grows. For instance, dichotomic insertion, which is only marginally slower than quick sort on 100 items, becomes 10 times slower on 10000 items. 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.**

Keywords: **Algorithm analysis, Sorting algorithm, Empirical Analysis Computational Complexity notations.**

I. INTRODUCTION

"Before there were computers, there were algorithms." But now that there are computers, there are even more algorithms, and algorithms lie at the heart of computing. What are algorithms? Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output. We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship. For example, we might need to sort a sequence of numbers into non decreasing order. This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools.

This paper is about algorithm theory introduction and just overview of algorithmic basics. If computers were infinitely fast, any correct method for solving a problem would do. You would probably want your implementation to be within the bounds of good software engineering practice (for example, your implementation should be well designed and documented), but you would most often use whichever method was the easiest to implement. Of course, computers may be fast, but they are not infinitely fast. And memory may be inexpensive, but it is not free. Computing time is therefore a bounded resource, and so is space in memory. You should use these resources wisely, and algorithms that are efficient in terms of time or space will help you do so.

II. COMPLEXITY OF ALGORITHM

The complexity of an algorithm is the cost, measured in running time, or storage, or whatever units are relevant, of using the algorithm to solve one of those problems.

T(n)	**Name**	**Problems**
O(1)	Constant	Easy-solved
O(log n)	Logarithmic	
O(n)	Linear	
O(n log)	Linear-log.	
$O(n^2)$	Quadratic	
$O(n^3)$	Cubic	
$O(2^n)$	Exponential	Hard-solved
O(n!)	Factorial	

III. 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.

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,

$T(n) = 2.t(n/2) + c.n$
$= 2.(2.t(n/4) + c.n/2) + c.n$
$= 2.(2.(2.t(n/8) + c.n/4) + c.n/2) + c.n$
$= 8.t(n/8) + 3.c.n$

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 _ (n^2). The Ideal behavior is O (*n*). Sort algorithms which only use an abstract key comparison operation always need (*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.

IV. ANALYSIS OF SORTING ALGORITHMS

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) Selection Sort

Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array. Selecting the lowest element requires scanning all n elements (this takes $n-1$ comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining $n - 1$ elements and so on, for $(n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1) / 2 \in \Theta(n^2)$ comparisons (see arithmetic progression). Each of these scans requires one swap for $n - 1$ elements (the final element is already in place). Among simple average-case $\Theta(n^2)$ algorithms, selection sort almost always outperforms bubble sort and gnome sort, but is generally outperformed by insertion sort. Insertion sort is very similar in that after the k^{th} iteration, the first k elements in the array are in sorted order. Insertion sort's advantage is that it only scans as many elements as it needs in order to place the $k + 1$st element, while selection sort must scan all remaining elements to find the $k + 1$st element. Simple calculation shows that insertion sort will therefore usually perform about half as many comparisons as selection sort, although it can perform just as many or far fewer depending on the order the array was in prior to sorting. It can be seen as an advantage for some real-time applications that selection sort will perform identically regardless of the order of the array, while insertion sort's running time can vary considerably. However, this is more often an advantage for insertion sort in that it runs much more efficiently if the array is already sorted or "close to sorted." While selection sort is preferable to insertion sort in terms of number of writes ($\Theta(n)$ swaps versus $O(n^2)$ swaps), it almost always far exceeds (and never beats) the number of writes that cycle sort makes, as cycle sort is theoretically optimal in the number of writes. This can be important if writes are significantly more expensive than reads, such as with EEPROM or Flash memory, where every write lessens the lifespan of the memory.

(B) 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 (n2) 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.

(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 (n2). 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) Quick Sort

From the initial description it's not obvious that quick sort takes $O(n \log n)$ time on average. It's not hard to see that the partition operation, which simply loops over the elements of the array once, uses $O(n)$ time. In versions that perform concatenation, this operation is also $O(n)$.

In the best case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size. Consequently, we can make only $\log n$ nested calls before we reach a list of size 1. This means that the depth of the call tree is $\log n$. But no two calls at the same level of the call tree process the same part of the original list; thus, each level of calls needs only *O*(*n*) time all together (each call has some constant overhead, but since there are only *O*(*n*) calls at each level, this is subsumed in the *O*(*n*) factor). The result is that the algorithm uses only *O*(*n* log *n*) time.

An alternative approach is to set up a recurrence relation for the *T*(*n*) factor, the time needed to sort a list of size n. Because a single quick sort call involves *O*(*n*) factor work plus two recursive calls on lists of size $n/2$in the best case, the relation would be.

$$T(n) = O(n) + 2T\left(\frac{n}{2}\right).$$

The master theorem tells us that *T*(*n*) = *O*(*n* log *n*).

In fact, it's not necessary to divide the list this precisely; even if each pivot splits the elements with 99% on one side and 1% on the other (or any other fixed fraction), the call depth is still limited to, so the total running time is still *O*(*n* log *n*).

V. 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.

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.