International Journal of Intelligent Information Processing
2(2) July-December 2008; pp. 363-368

SELECTION OF BEST SORTING ALGORITHM

ADITYA DEV MISHRA[*] & DEEPAK GARG[**]

The problem of sorting is a problem that arises frequently in computer programming. Many different sorting algorithms have been developed and improved to make sorting fast. As a measure of performance mainly the average number of operations or the average execution times of these algorithms have been investigated and compared. There is no one sorting method that is best for every situation. Some of the factors to be considered in choosing a sorting algorithm include the size of the list to be sorted, the programming effort, the number of words of main memory available, the size of disk or tape units, the extent to which the list is already ordered, and the distribution of values.

Keywords: *Sorting, complexity lists, comparisons, movement sorting algorithms, methods, stable, unstable, internal sorting*

1. INTRODUCTION

Sorting is one of the most important and well-studied problems in computer science. Many good algorithms are known which offer various trade-offs in efficiency, simplicity, memory use, and other factors. However, these algorithms do not take into account features of modern computer architectures that significantly influence performance. A large number of sorting algorithms have been proposed and their asymptotic complexity, in terms of the number of comparisons or number of iterations, has been carefully analyzed [1]. In the recent past, there has been a growing interest on improvements to sorting algorithms that do not affect their asymptotic complexity but nevertheless improve performance by enhancing data locality [3,6].

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. It is not always possible to say that one sorting algorithm is better than another sorting algorithm. Performance of various sorting algorithm depend upon the data being sorted. Sorting is generally understood to be the process of re-arranging a given set of objects in a specific order. In particular, sorting is an ideal subject to demonstrate a great diversity of algorithms, all having the same purpose, many of them being optimal in some sense and most of them having advantages over others. The example of sorting is moreover well-suited for showing how a significant gain in performance may be obtained by the development of sophisticated algorithm when obvious methods are readily available.

Many different sorting algorithms have been invented and paper will discuss about some of them in this paper. Why are there so many sorting methods? For computer science, this is a special case of question, "why there are so many x methods?", where *x*

* Computer Science and Engineering Department, Thapar University, Patiala.

ranges over the set of problem; and the answer is that each method has its own advantages and disadvantages, so that it outperforms the others on the same configurations of data and hardware. Unfortunately, there is no known "best" way to sort; there are many best methods, depending on what is to be sorted on what machine for what purpose. There are many fundamental and advance sorting algorithms. All sorting algorithm are problem specific means they work well on some specific problem, not all the problems. All sorting algorithm apply to specific kind of problems. Some sorting algorithm apply to small number of elements, some sorting algorithm suitable for floating point numbers, some are fit for specific range like (0 1]. some sorting algorithm are used for large number of data, some are used for data with duplicate values. We sort data either in numerical order or lexicographical order. We sort numerical value either in increasing order or decreasing order. In the paper, properties of the sorting techniques are summarized in the tabular form in the 4th section *"Comparison of various Sorting algorithms"*. The reader with a particular problem in mind may consult that section in order to decide which technique he should study in details.

2. OPTIMUM SORTING ALGORITHM

In comparing internal sorting algorithms to determine the best one for a number of elements, we are immediately confronted with the problems of defining what we mean by best sorting algorithm. First, we will define best sorting algorithm. Although there is no "best" definition of a best sorting algorithm, we will, nevertheless, define the best sorting algorithm as the one whose weighted sum of the number of comparisons, moves, and exchanges is minimal [1]. Indeed, all sorting algorithms are problem specifics means they do well on special kind of problems like some are useful for small number elements, some are for large list, some are suitable for repeated values. So, it's a difficult task to say that which sorting algorithm is best. But we demonstrate in this paper some internal sorting algorithms and compared them. In this paper we realize the shortcomings of our definition, but feel that its simplicity provides both an honest basis of comparison and insight into the nature of the algorithm. We feel that our measure is a better reflection of the sorting effort than the commonly used number of comparisons. Studies of the inherent complexity of sorting have usually been directed towards minimizing the number of times We make comparisons between keys while sorting n items, or merging m items with n or selecting the i^{th} largest of an unordered set of n items. To determine the best sorting algorithm we needed minimum-comparison sorting, minimum-comparison selection and minimum–comparison merging. The minimum number of key comparison needed to sort n elements is obviously zero, because we already known that there is no comparison in radix sort. We have also seen several sorting methods which are based essentially on comparisons of keys. Therefore it is clear that comparison counting is not the only way to measure the effectiveness of a sorting method. Minimum-Comparison Selection type of problem arises when we search for the best procedure to select the i^{th} largest of n elements.

3. COMPARISON OF ARRAY SORTING METHODS

To determine the good measure of efficiency among many possible solutions is obtained by counting the number C of needed key comparison and M of moves (transposition) of items. These numbers are function of the number n of item to be sorted.

In this section, compare straight methods accordingly their effectiveness because straight methods are particularly well-suited for elucidating the characteristics of the major sorting principles. In table 1, n denotes the number of item to be sorted, C and M Shall again stand for the number of required key comparisons and item moves, respectively. The column indicators min, avg, max specify the respective minima, maxima and expected values average over all $n!$ permutations of n item. No reasonably simple accurate formulas are available on the advanced methods. The essential facts are that the computational effort needed is $c_1 n^{1.2}$ in the case of shell sort and is $c_i.n.\log$ (n) in the case of heap sort and quick sort. These formulas merely provide a rough measure of performance as function of n, and they allow the classification of sorting algorithms into primitive, straight methods (n^2) and advanced or logarithmic methods ($n.\log\ n$). We would probably say it depends on how and why this sort will be implemented. While quick sort may be the fastest sorting algorithm, it isn't always the most efficient, and merge sort is extremely useful in online sorting. While insertion sort may be simple to implement, it is not more efficient than quick or merge sorts. It all depends on what you need the sort for.

4. COMPARISON OF VARIOUS SORTING ALGORITHMS

In this section, compare sorting algorithms according to their complexity, method used by them and also mention their advantages and disadvantages. In the following table, n represent the number of elements to be sorted. The column Average and worst case give the time complexity in each case. These all are comparison sort.

Table 1
Comparison of Comparison Based Sort

Name	*Average Case*	*Worst Case*	*Method*	*Advantage/Disadvantage*
Bubble Sort	O(n^2)	O(n^2)	Exchange	1. Straightforward, simple and slow. 2. Stable. 3. Inefficient on large tables.
Insertion Sort	O(n^2)	O(n^2)	Insertion	1. Efficient for small list and mostly sorted list. 2. Sort big array slowly. 3. Save memory
Selection Sort	O(n^2)	O(n^2)	Selection	1. Improves the performance of bubble sort and also slow. 2. Unstable but can be implemented as a stable sort. 3. Quite slow for large amount of data.
Heap Sort	O(n log n)	O(n log n)	Selection	1. More efficient version of selection sort. 2. No need extra buffer. 3. Its does not require recursion. 4. Slower than Quick and Merge sorts.
Merge Sort	O(n log n)	O(n log n)	Merge	1. Well for very large list, stable sort. 2. A fast recursive sorting. 3. Both useful for internal and external sorting.

Contd.

Name	*Average Case*	*Worst Case*	*Method*	*Advantage / Disadvantage*
				4. It requires an auxiliary array that is as large as the original array to be sorted.
In place-merge Sort	O(n log n)	O(n log n)	Merge	1. Unstable sort. 2. Slower than heap sort. 3. Needs only a constant amount of extra space in addition to that needed to store keys.
Shell Sort	O(n log n)	O(n log^2 n)	Insertion	1. Efficient for large list. 2. It requires relative small amount of memory, extension of insertion sort. 3. Fastest algorithm for small list of elements. 4. More constraints, not stable.
Cocktail Sort	O(n^2)	O(n^2)	Exchange	1. Stable sort. 2. Variation of bubble sort. 3. Bidirectional bubble sort.
Quick Sort	O(n log n)	O(n^2)	Partition	1. Fastest sorting algorithm in practice but sometime Unbalanced partition can lead to very slow sort. 2. Available in many slandered libraries. 3. O (log n) space usage. 4. Unstable sort and complex for choosing a good pivot element.
Library sort	O(n log n)	O(n^2)	Insertion	1. Stable 2. It requires extra space for its gaps.
Gnome sort	O(n^2)	O(n^2)	Exchange	1. Stable 2. Tiny code size. 3. Similar to insertion sort.

The following table described sorting algorithm which are not comparison sort. Complexities below are in terms of n, the number of item to be sorted, k the size of each key and s is the chunk size use by implementation. Some of them are based on the assumption that the key size is large enough that all entries have unique key values, and hence that $n << 2^k$.

Table 2
Comparison of Non-Comparison Sort

Name	*Average Case*	*Worst Case*	$n << 2^K$	*Advantage / disadvantage*
Bucket Sort	$O(n.k)$	O($n^2.k$)	No	1. Stable, fast. 2. Valid only in range o to some maximum value M. 3. Used in special cases when thekey can be used to calculate the address of buckets.
Counting Sort	$O(n + 2^k)$	$O(n + 2^k)$	Yes	1. Stable, used for repeated value. 2. Often used as a subroutine in radix sort. 3. Valid in the rang (o k]
Radix Sort	$O(n.\frac{k}{s})$	$O(n.\frac{k}{s})$	No	Where k is some integer. 1. Stable, straight forward code. 2. Used by the card-sorting machines. 3. Used to sort records that are keyed by multiple fields like date (Year, month and day).

Contd.

Name	*Average Case*	*Worst Case*	$n << 2^K$	*Advantage / disadvantage*
MSD Radix Sort	$O(n.\frac{k}{s})$	$O(n.\frac{k}{s})$	No	1. Enhance the radix sorting methods, unstable sorting. 2. To sort large computer files very efficiently without the risk of overflowing allocated storage space. 3. Bad worst-case performance due to fragmentation of the data into many small sub lists. 4. It inspects only the significant characters.
LSD Radix Sort	$O(n.\frac{k}{s})$	$O(n.\frac{k}{s}.2s)$	No	1. It inspects a complete horizontal strip at a time. 2. It inspects all characters of the input. 3. Stable sorting method.

5. PROBLEM DEFINITION AND SORTING METHODS

All the sorting algorithms are problem specific. Each sorting algorithms work well on specific kind of problems. In this section we described some problems and analyses that which sorting algorithm is more suitable for that problem.

Table 3
Sorting Algorithms According to Problem

Problem Definition	**Sorting Algorithms**
The data to be sorted is small enough to fit into a processor's main memory and can be randomly accessed i.e. no extra space is required to sort the records (Internal Sorting).	Insertion Sort, Selection Sort, Bubble Sort
Source data and final result are both sorted in hard disks (too large to fit into main memory), data is brought into memory for processing a portion at a time and can be accessed sequentially only (External Sorting).	Merge Sort (business application, database application)
The input elements are uniformly distributed within the range [0, 1).	Bucket Sort
Constant alphabet (ordered alphabet of constant size, multiset of characters can be stored in linear time), sort records that are of multiple fields.	Radix Sort
The input elements are small.	Insertion Sort
The input elements are too large.	Merge Sort, Shell Sort, Quick Sort, Heap Sort
The data available in the input list are repeated more times i.e. occurring of one value more times in the list.	Counting Sort
The input elements are sorted according to address (Address Computation).	Proxmap Sort, Bucket Sort
The input elements are repeated in the list and sorted the list in order to maintain the relative order of record with equal keys.	Bubble Sort, Merge Sort, Counting Sort, Insertion Sort
The input elements are repeated in the list and sorted the list so that their relative order are not maintain with equal keys.	Quick Sort, Heap Sort, Selection Sort, Shell Sort
A sequence of values, $a_0, a_1, \ldots, a_{n-1}$, such that there exists an i, $0 \le i \le n-1$, a_0 to a_i is monotonically increasing and a_i to a_{n-1} is monotonically decreasing. (sorting network)	Biotonic-merge Sort

CONCLUSION

In this paper, try to summarize nearly all the sorting algorithms. Therefore, to sort a list of elements, First of all we analyzed the given problem i.e. the given problem is of which type (small numbers, large values). After that we apply the sorting algorithms but keep in mind minimum complexity, minimum comparison and maximum speed. In this paper we also discuss the advantages and disadvantages of sorting techniques to choose the best sorting algorithms for a given problem. Finally, the reader with a particular problem in mind can choose the best sorting algorithm.

REFERENCES

[1] Knuth, D. The Art of Computer Programming Sorting and Searching, 2nd edition, **3**. Addison-Wesley, (1998).

[2] Wirth, N., 1976. Algorithms + Data Structures = Programs: Prentice-Hall, Inc. Englewood Cliffs, N.J.K. Mehlhorn. Sorting and Searching. Springer Verlag, Berlin, (1984).

[3] LaMarca and R. Ladner. The Influence of Caches on the Performance of Sorting. In Proceeding of the *ACM/SIAM Symposium on Discrete Algorithms*, (January 1997), 370–379.

[4] Hoare, C.A.R. Algorithm 64: *Quicksort. Comm*. ACM **4**, 7 (July 1961), 321.

[5] G. Franceschini and V. Geffert. An In-Place Sorting with O (n log n) Comparisons and O (n) Moves. In Proc. *44th Annual IEEE Symposium on Foundations of Computer Science*, (2003), 242–250.

[6] D. Jim´enez-Gonz´alez, J. Navarro, and J. Larriba-Pey. CC-Radix: A Cache Conscious Sorting Based on Radix Sort. In *Euromicro Conference on Parallel Distributed and Network based Processing*, (February 2003), 101–108.

[7 J. L. Bentley and R. Sedgewick.Fast Algorithms for Sorting and Searching Strings. *ACM-SIAM SODA* '97, (1997), 360–369.

[8] Flores, I. Analysis of Internal Computer Sorting. *J.ACM* **7**, 4 (Oct. 1960), 389–409.

[9] Flores, I. Analysis of Internal Computer Sorting. *J.ACM* **8**, (Jan. 1961), 41–80.

[10] Andersson, A. and Nilsson, S. 1994. A New Efficient Radix Sort. In *Proceeding of the 35th Annual IEEE Symposium on Foundation of Computer Science* (1994), 714–721.

[11] Davis, I. J. A Fast Radixsort. *The Computer Journal* **35**, (6), (1992), 636–642.

[12] V. Estivill-Castro and D.Wood. A Survey of Adaptive Sorting Algorithms. *Computing Surveys*, **24**, (1992), 441–476.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd edition, 2001.