RESEARCH ARTICLE OPEN ACCESS

Basics of Algorithm Selection : A Review

Niharika Singh [1], DR. R. G. Tiwari [2]
Department of Computer Science and Engineering
SRMGPC, Lucknow
India

ABSTRACT

Sorting is the most basic activity in computational world. It is helpful in analysing data from various perspectives. Multiple sorting alternatives exist today. Each have their own advantages and disadvantages. We can not say that a particular algorithm is best. For different data patterns different algorithms can be implemented to achieve better results in minimum time and with limited memory space. In this paper we would compare various sorting algorithms on the basis of complexities. In the section 1, we have introduced sorting. In section 2 and 3, we have discussed complexity with its type and best – Average case complexities. In the next section we compare different algorithms on the basis of complexities and try to find the best one

Keywords :-Sorting, Analyses, Complexity, Performance.

I. INTRODUCTION

Sorting is used in various computational tasks. Many different sorting algorithms have been introduced so far. For measuring performance, mainly the average number of operations or the average execution times is considered. There is no known "best" way to sort; there are many best methods, depending on what is to be sorted on what machine and 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. Some sorting algorithms apply to small number of elements, some sorting algorithm suitable for floating point numbers. Sorting algorithm depend on pattern of data values to be sorted.

It can not be necessarily stated that one algorithm is better than another, since relative performance can vary depending on the type of data being sorted. In some situations, most of the data are in the correct order, with only a few items needing to be sorted; In other situations the data are completely mixed up in a random order and in others the data will tend to be in reverse order. Different algorithms will perform differently according to the data being sorted. Performance [1] means how much time/memory/disk/... is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code. Complexity, how do the resource requirements of a program or algorithm scale, i.e., what happens, as the size of the problem being solved gets larger? Complexity affects performance but not the other way around. When we consider the complexity of a method, we don't really care about the exact number of operations that are performed; instead, we care about how the number of operations relates to the problem size. If the problem size doubles, does the number of operations stay the same? Double? Increase in some other way? For constant-time methods like the size method, doubling the problem size does not affect the number of operations.

II. ALGORITHM COMPLEXITY

Complexity [1] of the sorting algorithm can be expressed using asymptotic notation. These notations help us predict the best, average and poor behaviour of the sorting algorithm. The various notations are as follow [3]:

1. O(x) = Worst Case Running Time
2. Ω (x) = Best Case Running Time
3. Θ (x) = Best and Worst case are the same

A. Big-O Notation

A theoretical measure of the execution of an algorithm usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation f (n) = O (g(n)) means it is less than some constant multiple of g(n). The notation is read, "f of n is big oh of g of n".

Definition: f (n) = O (g (n)) means there are positive constants c and k, such that $0 \leq f(n) \leq cg(n)$ for all $n \geq k$. The values of c and k must be fixed for the function- f-and-must-not-depend-on-n.

B. Theta Notation (Θ)

A theoretical measure of the execution of an algorithm usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation f(n) = Θ (g(n)) means it is within a constant multiple of g(n). The equation is read, "f of n is theta g of n".

Definition: f(n) = Θ (g(n)) means there are positive constants c_1, c_2, and k, such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq k$. The values of c_1, c_2, and k must be fixed for the function f and must not depend on n[2].

C. Omega Notation (ω)

A theoretical measure of the execution of algorithms usually the time or memory needed, given the problem size n, which is usually the number of items. Informally, saying some equation f(n) = ω (g(n)) means g(n) becomes insignificant relative to f(n) as n goes to infinity.

Definition: f(n) = ω (g(n)) means that for any positive constant c, there exists a constant k, such that $0 \leq cg(n) < f(n)$ for all $n \geq k$. The value of k must not depend on n, but may depend on c[2].

III. BEST-CASE AND AVERAGE–CASE COMPLEXITY

Some methods may require different amounts of time on different calls, even when the problem size is the same for both calls. For example, we know that if add before is called with a sequence of length N, it may require time proportional to N (to move all of the items and/or to expand the array). This is what happens in the worst case. However, when the current item is the last item in the sequence, and the array is not full, add Before will only have to move one item, so in that case its time is independent of the length of the sequence; i.e., constant time. In general, we may want to consider the best and average time requirements of a method as well as its worst-case time requirements. Which is considered the most important will depend on several factors. For example, if a method is part of a time-critical system like one that controls an airplane, the worst-case[4] times are probably the most important (if the plane is flying towards a mountain and the controlling program can't make the next course correction until it has performed a computation, then the best-case and average- case times for that computation are not relevant - the computation needs to be guaranteed to be fast enough to finish before the plane hits the mountain).

On the other hand, if occasionally waiting a long time for an answer is merely inconvenient (as opposed to life-threatening), it may be better to use an algorithm with a slow worst-case time and a fast average-case time, rather than one with so-so times in both the average and worst cases.

For add Before, for a sequence of length N, the worst-case time is O(N)[5], the best-case time is O(1), and the average-case time (assuming that each item is equally likely to be the current item) is O(N), because on average, N/2 items will need to be moved.

Note that calculating the average-case time for a method can be tricky. You need to consider all possible values for the important factors, and whether they will be distributed evenly. Recall that when we use big-O notation, we drop constants and low-order terms. This is because when the problem size gets sufficiently large, those terms don't matter. However, this means that two algorithms can have the same big-O time complexity, even though one is always faster than the other. For example, suppose algorithm 1 requires N^2 time, and algorithm 2 requires $10 * N^2 + N$ time. For both algorithms, the time is $O(N^2)$, but algorithm 1 will always be faster than algorithm 2 [2]. In this case, the constants and low- order terms do matter in terms of which algorithm is actually faster.

However, it is important to note that constants do not matter in terms of the question of how an algorithm "scales" (i.e., how does the algorithm's time change when the problem size doubles). Although an algorithm that requires N^2 time will always be faster than an algorithm that requires $10*N^2$ time, for both algorithms, if the problem size doubles, the actual time will quadruple. When two algorithms have different big-O time complexity [3], the constants and low-order terms only matter when the problem size is small.

IV. ANALYSIS OF SORTING ALGORITHMS

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

In table 1, we see that a sorting algorithm depends upon the characteristics of problem. Like given list is sorted so that maintain the relative order of records with equal keys, repeated value occurs more times and the elements in the list are small ,counting sort is efficient. Quick sort and heap sort both are use for large number and also satisfy the property of unstable. The worst case running time for quick sort [5] is $O(n^2)$ which is unacceptable for large sets but typically faster due to better cache performance. Heap sort also competes with merge sort[5] ,which has the same time bound but requires O(n) auxiliary space, whereas heap sort requires only a constant amount. Heap sort [5] is unstable while quick sort is stable sort. Bucket sort is used when give list is sorted according to address and the elements of list are uniformly distributed over a range (01]. Insertion sort and selection sort both are internal sorting but one is stable and another is unstable. Therefore to chose a best sorting algorithm

first of all analyse the characteristics of problem.

TABLE I

Problem Definition and Sorting algorithms

Problem Definition	Sorting Algorithms
When sufficient memory is available and data to be sorted is small enough to fit into a processor's memory and no extra space is required to sort the data values..	Bubble sort, Insertion Sort, Selection Sort
When data size is too large to accommodate data into main memory	Merge Sort
The input elements are uniformly distributed within the range [0, 1].	Bucket Sort
Constant alphabet (ordered alphabet of constant size, multi set of characters can be stored in linear time), sort records that are of multiple fields.	Radix 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. Re-occurrence of value in the list.	Counting Sort
The input elements are sorted according to address	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, Insertion Sor
The input elements are repeated in the list and sorted the list so that their relative orders are not maintain with equal keys.	Quick Sort, Heap Sort, Selection Sort,
A sequence of values, a_0, $a_1 \ldots a_{n-1}$, such that there exists an i, $0 \leq i \leq n-1$, a_0 to a_i is monotonically increasing and a_i to a_{n-1} is monotonically decreasing. (sorting network)	Biotonic -merge Sort
Adaptive sorting algorithms that are comparison based and do not put any restriction on the type of keys, uses data structure like linked list, stack, queue.	Stack sort , Min-max sort

V. COMPARISON OF VARIOUS SORTING

In the following table, compare sorting algorithms according to their complexity, method used by them like exchange, insertion, selection, merge 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 2
Comparison of Comparison Based Sort

Name	Average Case	Worst Case	Method
Bubble Sort	$O(n^2)$	$O(n^2)$	Exchange
Insertion Sort	$O(n^2)$	$O(n^2)$	Insertion
Selection Sort	$O(n^2)$	$O(n^2)$	Selection
Heap Sort	O(n log n)	O(n log n)	Selection
Merge Sort	O(n log n)	O(n log n)	Merge
In place- merge Sort	O(n log n)	O(n log n)	Merge
Shell Sort	O(n log n)	O(nlog 2n)	Insertion
Quick Sort	O(n log n)	O(n2)	Partition

VI. COMPARISON OF NON COMPARISON BASED SORTING ALGORITHM

Table 3 describes sorting algorithm which are not based on 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 3
Comparison of non comparison based sorting algorithms

Name	Average Case	Worst Case	$n<<2^K$
Bucket Sort	O(n.k)	$O(n^2.k)$	No
Counting Sort	$O(n+2^k)$	$O(n+2^k)$	Yes
Radix Sort	O(n. k/s)	O(n. k/s)	No
MSD Radix Sort	O(n. k/s)	O(n. k/s)	No
LSD Radix Sort	O(n. k/s)	O(n. k/s)	No

VII. ADVANTAGES AND DISADVANTAGES

Every sorting algorithm has some advantages and disadvantages. In the following table we are tried to show the strengths and weakness of some sorting algorithms according to their order, memory used, stability, data type and complexity. To determine the good sorting algorithm ,speed is the top consideration but other factor include handling various data type, consistency of performance, length and complexity of code, and the prosperity of stability.

Table 4
Advantages and memory consumption in various Algorithms

sort	order	worst case	memory
quick sort	n log n	n^2	NK+NP+STACK
merge	n log n	n log n	NK+2NP+STACK
heap	n log n	n log n	NK+NP
Shell	n (log n)2	n	NK+NP
insertion	n^2	n^2	NK+NP
selecti	n^2	n^2	NK+NP

VIII. CONCLUSION

We have compared various algorithms popular for sorting data of varied type. We found that selection of algorithm should depend on specific type of data available. We need to analyse data then implement sorting according to the data. We have discussed strengths and shortcomings of all major algorithms.

REFERENCES

[1] Y. Han "Deterministic sorting in O (n log log n) time and linear space", Proceedings of the thirty-fourth annual ACM symposium on Theory of computing, Montreal, Quebec, Canada, 2002, p.602-608.Akhawe, D., Barth, A., Lam, P. E., Mitchell, J.C. and Song, D. "Towards a formal foundation of web security", CSF, pp. 290-304, 2010.

[2] M. Thorup "Randomized Sorting in O (n log log n) Time and Linear Space Using Addition, Shift, and Bit-wise Boolean Operations", Journal of Algorithms, Volume 42, Number 2, February 2002, p. 205-230 .

[3] Y. Han, M. Thorup, "Integer Sorting in O(n √(log log n) Time and Linear Space", Proceedings of the 43rd Symposium on Foundations of Computer Science, 2002, p. 135-144.

[4] P. M. McIlroy, K. Bostic and M. D. McIlroy, "Engineering radix sort", Computing Systems, 2004, p.224-230.

[5] M. D. McIlroy, "A killer adversary for quick sort", Software--Practice and Experience, 1999, p.123-145.

[6] http://209.85.141.104/search?q=cache:iBhfoE92zZoJ :www.softpanorama.org/Algorith ms/sorting.shtml+introduction+of+sorting+algorithm &hl=en&ct=clnk&cd=9&gl=in, accessed on 20 March, 2007.