International Journal of Information Sciences and Application.
ISSN 0974-2255 Volume 3, Number 1 (2011), pp. 25-30
© International Research Publication House
http://www.irphouse.com

An Efficient Quicksort using Value based Pivot Selection an Bidirectional Partitioning

Runumi Devi and Vineeta Khemchandani

Department of Computer Applications, JSS Academy of Technical Education, C-20/1Sector 62, Noida, Uttar Pradesh-201301, India
E-mail: runumi@jssaten.ac.in, vkemchandani@jssaten.ac.in

Abstract

Quick sort is generally considered to be the best internal sorting algorithm, and is often used as a yardstick by which the efficiency of other sorting algorithms is compared. It is, therefore essential that its performance is studied thoroughly. This includes studying the worst case behaviour of the algorithm, and especially when the algorithm is experimentally evaluated. The worst case running time of quick sort algorithm is O(n2). This paper proposes a new variant of quick sort, which reduces the running time of the algorithm to n log(n). Two versions of the quick sort are being studied – the Classical quick sort and the proposed one. A comparison is made in terms of running time which is further established by mathematical analysis. The simulation result shows that proposed algorithm is faster than Classical quick sort in the worst case for sorting the same input data size.

Keywords: Quick sort, sorting, asymptotic, algorithm, in-place sorting.

Introduction

Sorting data is one of the most fundamental problems in Computer Science, especially if the arranging objects are primitive ones, such as integers, bytes, floats, etc. Since sorting methods play an important role in the operations of computer and other data processing systems, there has been an interest in seeking new algorithms better than the existing ones.

Donald Knuth [3] reports that "computer manufacturers of the 1960s estimated that more than 25 percent of the running time on their computers was spent on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time.." As i expected, sorting is one of the most heavily studied problems

in computer science. Considerable effort has been made to design sorting algorithms with optimal asymptotic efficiency. Of course, the speed of sorting can depend quite heavily on the environment in which the sorting is done and the distribution of items.

Quick sort is considered the most efficient sorting algorithm as it is an in-place algorithm (even though it's a recursive algorithm, it uses a small auxiliary stack), and has an average sorting time of n log(n) to sort n items. The algorithm has been analyzed and studied extensively in [1], [5] and [6]. The main drawback of the algorithm is its worst case time complexity of $O(n^2)$, which occurs when the list of values is already sorted or nearly sorted, or sorted in reverse order [7]. Quicksort is a divide-and-conquer algorithm

Since its development in 1961 by Hoare, the Quick sort algorithm has experienced a series of modifications aimed at improving the O(n2) worst case behavior. The performance of quicksort[2] depends on the choice of pivot element as well as the sorting techniques that is being used to sort the sublist of trivial size. Moreover, performance also depends on the way partitioning is done. In this study, we propose a new variant of quick sort which considers the above mentioned criteria to reduce the worst case running time.

The rest of this paper is organized as follows. In section 2 we give a brief review of classical quick sort.Section3 introduces the proposed variant of quick sort. Section 4 discusses the performance analysis. Section5 presents the comparative experimental results. The paper concludes with section6.

Classical Quick sort

The classical approach of the Quick sort is based on choosing either the first or the last key as the pivot element. To sort a list of n values represented by a one dimensional array A indexed from 1 to n, this algorithm partitions the array into two parts, a left subarray and a right subarray. The keys in the array will be reordered such that all the elements in the left subarray are less than the pivot and all the elements in the right sub array are greater than the pivot. Then the algorithms proceeds to sort each subarray independently. Different choices for the pivot results in different variations of the Quick sort algorithm.

Proposed Algorithm

The proposed algorithm adopts different approach for pivot selection and partitioning such that pivot element always comes near the middle in a list of keys. Given an array of size n, the algorithm, recursively partitions the array into two sub-arrays till the array-size meets the trivial point (<=3). The approach consists of three phases, firstly pivot selection phase, second partition phase and finally sorting trivially sized sub-arrays phase. Based on the above concepts, we have the following algorithm for sorting a list

Algorithm

```
QUICKSORT (L, p, r)
```

```
1 size ← r-p+1
2 if size <= 3
3 then
4 MANUAL_SORT (L, p, r)
5 else
6 pivot←CALCULATE_PIVOT(L,p,r)
7 q ← PARTITION(L,p,r,pivot)
8 QUICKSORT (L, p, q)
9 QUICKSORT (L,q+1,r)
```

Pivot Selection

Given a list L[p..r] of keys of size n, in the first step, we found the middle location at n/2 and the list is divided into two sublist L[left] and L[right].Once the middle location is found, minimum and maximum value from the two sublists are calculated, where left<=n/2 and right > n/2

The pivot value is then found as a mean of four values which are, min(L[left]), max(L[left]), min(L[right]) and max(L[right]).

pivot = (min(L[left])+ max(L[left])+ min(L[right])+ max(L[right]))/4.

Partitioning

The key to the algorithm is the PARTITION procedure, which rearranges the array L in-place. The PARTITION procedure selects a pivot element around which the partition of the sub array L[p---r] has to be done. It uses the HOARE type partitioning, which divides the list in two partitions L[p..i] and L [i+1..r].

```
PARTITION ( L, p, r, pivot)
 1 i =l; j=r;
 2        while TRUE
 3 do
 4 while L[i] <= pivot
 5 do       i++
 6        while L[j] >= pivot
 7 do j--
 8         if i>=j
 9  then
10        break
11  else
12  exchange L[i] ↔L[j]
```

This procedure is executed on both the sub arrays L[left] and L[right] simultaneously. In each iteration of the partition, elements of the two subarrays are compared with the pivot element. There are two cases to be considered depending on the outcome of the tests in line 4 and line 6, which are when L[i] <= pivot and when L[j] >= pivot. When L[i] <= pivot, the only action in the loop is to increment i and

when L[j] >= pivot, the only action is to decrement j. After the termination of the loops at line 4 and line 6, L[i] and L[j] are swapped.

The procedure terminates at i=j, This means that all the elements on the left and on the right subarrays have been visited and compared with. This condition is important to ensure the correctness of the algorithm.

Manual Sorting

Manual sorting is done once the array size reaches the trivial point. Insertion sort can be one of the optional choice to sort small size array.

```
MANUAL_SORT(L, p, r)

1size ← r -l +1
2 if size <= 1
3 then
4 return;
5 if size == 2
6 then
7 if L[l] > L[r]
8 then
9 exchange L[l]↔L[r]
10 if size ==3
11 then
12 if L[l]>L[r-1]
13 then
14 exchange L[l] ↔ L[r-1]
15 if(L[l] > L[r]
16 then
17 exchange L[l] ↔ L[r]
18 if L[r-1] > L[r]
19 exchange L[r-1]↔L[r]
```

Performance analysis of the proposed algorithm

The proposed algorithm gives better running time than classical quick sort algorithm as it is based on mean calculation for selection of the pivot element. Here, arithmetic mean is calculated on four extreme values i.e two minimum extreme and two maximum extreme values. This pivot selection procedure is repeated for each iteration of the quick sort till the trivial case is achieved. Thus, in this scenario, there will not be a situation which will lead to a worst case partitioning. So, the partitioning can always claimed to be balanced.

The partitioning requires at most n comparisons, which costs θ (n) time. Since the partitioning procedure chooses mean of extreme values as pivot[4], and mean always comes between extreme values, so, partitioning splits the list into 8-to-2. Thus, the recurrence for the proposed algorithm can be represented as

$$T(n) = T(8n/10) + T(2n/10) + c\,n$$

The recurrence terminates at the boundary condition log10/8(n). The total cost of the quick sort is therefore, O (nlgn). With a 8-to-2 proportional split at every level of recursion, which intuitively seems quite unbalanced, this proposed variants of quicksort, runs in O(nlgn, which is same as if the split were right down the middle.

Experimental Evaluation

In this section, we present a performance comparison of our development with the Classical quick sort algorithm. To study the effectiveness of our algorithm we have used turbo C++ to implement both the versions of the algorithm.

The experiments were conducted on a computer with an Intel Pentium 4 processor with a speed of 2.66 MHz, and 512 MB of RAM. The sizes of the arrays ranged from n= 500 to n = 500,000 elements. Both the algorithms are implemented on the same machine for the same input data. Table1 summarizes and provides a means to briefly compare the two algorithms. Our simulation results show that the performance of proposed algorithm is better than quick sort in term of running time (Table 1).

Table 1: Comparisons of algorithms.

Input Size (n)	Classical Quick sort	Proposed Algorithm
500	0.241	0.193
1000	1.64	0.923
5000	3.313	2.245
10000	20.478	13.321
50000	40.456	28.093

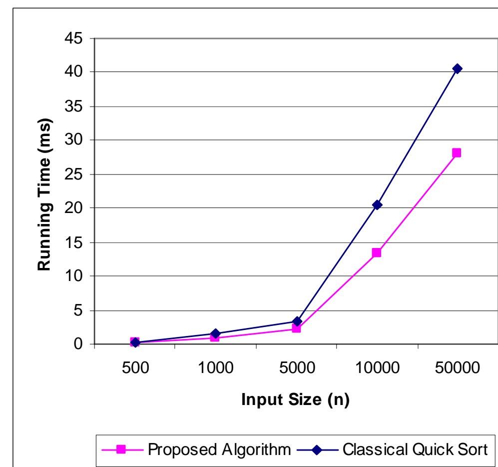

Figure 2: Running time of both classical Quick sort and proposed algorithms.

The performance of the proposed algorithm and the Classical quick sort are shown in figure2. The X-axis in this graph represents the input size and Y-axis represents the running time (ms). In the graph, we see that, for smaller input size (1000-5000) the running time of our algorithm is approximately same as classical quick sort. However, for larger input size (above 10000) the algorithm outperforms the classical quick sort algorithm.

Conclusion and Future Work

This paper presented a comparative performance of the proposed algorithm with respect to classical quick sort. It is found that the worst case running time of the algorithm is of the order of $n \log(n)$. This reduction has been possible because partitioning is done simultaneously from both directions and pivot selection is based on mean of four extreme values. Since the partitioning is happening simultaneously, the algorithm is likely to give better results if this operation is performed in parallel environment.

References

[1] Chaudhuri, R.; Dempster, A. C.(1993): A note on slowing Quicksort, SIGCSE vol.25, no. 2.

[2] Khreisat, L.(2007): QuickSort A Historical Perspective and Empirical Study, IJCSNS International Journal of Computer Science and Network Security, vol.7, no.12.

[3] Knuth, D.E.(1998):The Art of Computer Programming, 2nd ed, vol. 3, sorting and searching, Addison Wesley Publishing Co., Inc., Redwood City, CA.

[4] Leon, M.; Zawojewski, J.(1990): Use of the Arithmetic Mean: An Investigation of Four Properties Issues and Preliminary Results, International Conference on the Teaching of Statistics, ICOTS 3.

[5] Sedgewick, R.(1997):The Analysis of Quicksort Programs, ActaInformatica 7, pp 327 – 355.

[6] Sedgewick, R.(1978): Implementing Quicksort programs, Comm. of ACM, 21(10), pp 847 – 857.

[7] Wainwright, R. L.(1985): A class of sorting algorithms based on Quicksort, Comm. ACM, vol. 28 no. 4.