INTERNATIONAL JOURNAL OF ADVANCE RESEARCH AND DEVELOPMENT

(Volume3, Issue7)

Available online at: www.ijarnd.com

QSort– Dynamic pivot in original Quick Sort

Nisha Rathi

Assistant Professor, Acropolis Institute of Technology & Research, Indore, Madhya Pradesh

ABSTRACT

This Paper proposes an efficient Quick Sort algorithm for sorting a large array. Quick Sort has proved to be the fastest sorting algorithm used for sorting, making O (n log n) comparisons to sort n items. The proposed algorithm is based on a dynamic pivot selection to enhance the average case and eliminate the worst case behavior of the original Quick Sort. The suggested QSort is data-dependent to increase the chances of splitting the array into relatively equal sizes. The algorithm is smart enough to recognize a sorted array that doesn't require further processing.

Keywords— *Dynamic pivot selection, Quick sort, QSort, Median-of-Three rule, Median-of-Five rule*

1. INTRODUCTION

Hoare [1] in 1961 first introduced the Quick Sort. Several alternatives have been developed and many of them are amongst the available fastest generic sorting algorithms. Quick Sort is the best option as the sorting algorithm. Quick Sort is based on the technique of divide and conquers. It recursively split each array into two sub-arrays. This splitting makes it easier to solve smaller problems as compared to a single larger one (Dean [6] in 2006, Ledley[7] in 1962). In Quick Sort, from the unsorted array, a pivot is selected which is used to divide the array into left and right sub-arrays. The same algorithm is called recursively until all sub-arrays have size one or zero. An average runtime complexity of the Quick Sort algorithm is Θ (n log n). When processing an already sorted list and picking the largest element as a pivot, the worst case complexity of Quick Sort algorithm Θ (n^2)

The Quick Sort algorithm's runtime depends on the splitting of the array and the consecutive sub-arrays. If splitting constantly results in a small reduction in the size of the array or sub-array, the resultant runtime will be T (n) = n + T (n-c), where c is a constant. This recurrence relation results to T (n) = Θ (n^2).If splitting of array results in almost equal size sub-arrays, the runtime complexity of Quick Sort will be reduced to T(n) = n + T(n/2). Thereby results to logarithmic time complexity T (n) = n + T (n/2) i.e. T (n) = Θ (n log n).

The finest performance of the Quick Sort algorithm came by splitting the array into almost equal size sub-arrays. These almost equal halves reduce the number of recursive calls and eventually reduce the execution time.

2. ENHANCEMENTS ON BASIC QUICK SORT ALGORITHM

Basic Quick Sort algorithm by Hoare [1] has gone through two main enhancements:

a) When the size of the input becomes relatively small and only a few elements are out of order then stops the recursive calling to Quick Sort. For such a case, the better choice is to use Insertion sort for getting linear time performance for almost ordered sub-arrays (Bell [7] in 1958), Box [8] in 1991), Kruse [9] in 1999).

b) The subsequent step up is based on the technique of pivot selection which is the most crucial factor in dividing of the array into sub-arrays. The variants of Quick Sort based on pivot selection techniques are as follows:

i. The left-most or the right-most element, chosen as a pivot in the original Quick Sort algorithm, causes the worst case behavior when sorting a sorted or almost sorted list.

ii. A different approach in pivot selection i.e. Random pivot selection technique reduces the likelihood of happening for the worst case situation. The Median-of-Three splitting technique (Sedgewick [10] in 1978) suggests to pick the pivot as the median of the values stored in the first, last and ((first + last) /2) indexes. Random pivot selection decreases the probability of the worst-case scenario and strengthens the chances of the average case performance of the algorithm. Though it does not assure the division of array into equal halves and may result in the worst case behavior of Quick Sort, even though it's not likely to take place.

iii. The Median-of-Five with random index selection technique (Janez [11] in 2000) is a modification for the technique mentioned in (ii) by adding together the values stored in two randomly selected indexes to the values stored in the Beg, End and ((Beg + End) /2) indexes. Even if this technique may provide a more balanced division of array as compared to previous ones, it can still go through the same difficulty of the previous techniques.

iv. To decrease the overhead related with the random number generation, the pivot is chosen as the median of the five values stored in fixed indexes i.e. Beg, ((Beg + End) /4), ((Beg + End) /2), (3 *(Beg + End) /4) and End (Mohammed [2] in 2004). To pick the median of five elements at each recursive call this technique still requires notable time.

v. Median-of-Seven and Median-of-Nine either with or without random index selection were proposed by Mohammed [2] in 2004. A more balanced split results on increasing the number of elements but it increases the time needed to pick the pivot at each recursive call. The selection of the pivot is dependent on a definite number of elements which does not essentially represent the nature of the array.

The worst case behavior of the Quick Sort algorithm may possibly still arise when using any of the above "pivot selection" techniques.

This paper is well thought-out in Section 3 which presents the Advanced QSort algorithm using the dynamic pivot selection technique, in Sections 4 we discuss the working of QSort and Section 5 presents the empirical analysis of QSort in comparison with the original Quick Sort and the Median of three methods. This paper brings to a close with section 6.

3. DYNAMIC PIVOT SELECTION TECHNIQUE

Keeping in consideration the value of each element in the array, the proposed pivot-selection technique is based on dividing the array into almost equal halves so that for each recursive call, there will be a reduction in the number of recursive calls and the overall execution time of Quick Sort. The projected approach is intended to ensure identical splitting for the array and is, therefore, makes the worst case performance to be O (n log n). If the array is already sorted, the algorithm will not be processed any further which reduces the O (n2) complexity into the best case behavior of the algorithm; i.e. O (n).

This Advanced approach will operate as follows; at first, the pivot value is chosen to be the value of the rightmost element of the array. There will be the comparison of every element value with the pivot. The counters namely C1 and C2 are utilized to count the number of elements with values smaller than the pivot versus the number of elements with values larger than the pivot. Similarly, S1 and S2 are utilized to store the sum of the values of the elements smaller than the pivot and the sum of those larger than the pivot. These variables are then used to calculate the next pivots for the recursive calls. In the recursive call for the left sub-array, the integer average of the values smaller than the pivot is passed as the pivot value. Likewise, in the recursive call for the right sub-array, the integer average of the values larger than the pivot is passed as the pivot value.
This advanced technique for picking pivot helps in consecutively dividing the array into almost equal halves thus improves the efficiency of the Quick Sort algorithm. To reduces the number of recursive calls, a Boolean variable flag is utilized by the algorithm to recognize an already sorted array or sub-array

Along with the reduction in recursive calls, the proposed technique converts the worst-case scenario for the classical Quick Sort algorithm into a best-case scenario with Θ (n) runtime. The modified code for QSort is provided below:

```
void Qsort(int a[],int beg,int end, int pivot)
 {
 int i=beg, j=end, flag=0,p1=0,p2=0,temp,c1=0,s1=0,c2=0,s2=0,z;
        if(beg<end)
         {
        temp=a[end];
        while(i<=j)
        {
                if(a[i]<=pivot)
                {
                        c1++; s1+=a[i];
                        if(flag==0 && temp>=(pivot-a[i])) temp=pivot-a[i];
                        else flag=1;
                                i++;
                }
                else
                {
                c2++;  s2+=a[i];
                z=a[i];
                a[i]=a[j];
                a[j]=z;
                j--;
                }
        }
        if(c1!=0)
                {
                p1=s1/c1;
                if(flag!=0) QS(a,beg,i-1, p1);
                }

        if(c2!=0)
                {
                p2=s2/c2;
                Qsort (a,i,end, p2);
                }

        }
}
```

Fig. 1: The QSort Algorithm

4. WORKING OF QSORT

In the QSort algorithm, the contents of the rightmost section of the array will be chosen as a pivot. In order to stop recursive calls while calculating the mean of the elements less than the pivot and that of the elements larger than the pivot, the proposed QSort algorithm makes use of a Boolean variable flag which will have a TRUE value if the array is sorted.

Therefore, for a sorted array, only one recursive call is required for the first and only required iteration with Θ (n) runtime requirement. Similar working will be there for sorted sub-arrays thus leads to reduce the execution time.

The QSort algorithm is likely to compute the requisite pivots for the subsequent two recursive calls in the current iteration for dividing the array into approximately two equal sub-arrays for an average case. Thus average case runtime requirements will be expressed as T (n) = n + T ((n/2) + c) + T ((n/2) -c), where c is a constant, n component is the time for calculating the pivot for the next iteration, and T ((n/2) + c) and T ((n/2)-c) are the times desirable for recursively sorting the two almost equal sub-arrays. Solving T (n) is resulting into Θ (n log n) as a time requirement.

When there are just a few elements of the array with extreme values, the worst case situation of the QSort algorithm will take place. The algorithm is likely to take apart the extreme values in early iterations and then carry on with the remaining elements with recursive calls which will strictly have an average case behavior wit**h Θ** (n log n) runtime requirement

5. EMPIRICAL ANALYSIS

Table1: Assessment of time (in seconds) for three algorithms requited to execute unsorted arrays with varying sizes

No. of elements in the array (unsorted)	Original Quick Sort (Pivot as the last element)	Median of three Method	QSort
100	0.01532	0.04658	0.01475
200	0.03948	0.01611	0.009376
300	0.05855	0.08117	0.03299
400	0.01427	0.06036	0.01155
500	0.01621	0.02082	0.01194
1000	0.03514	0.03975	0.05433
2000	0.05783	0.06043	0.03083
5000	0.07183	0.04501	0.04339
10000	0.06299	0.06211	0.04447

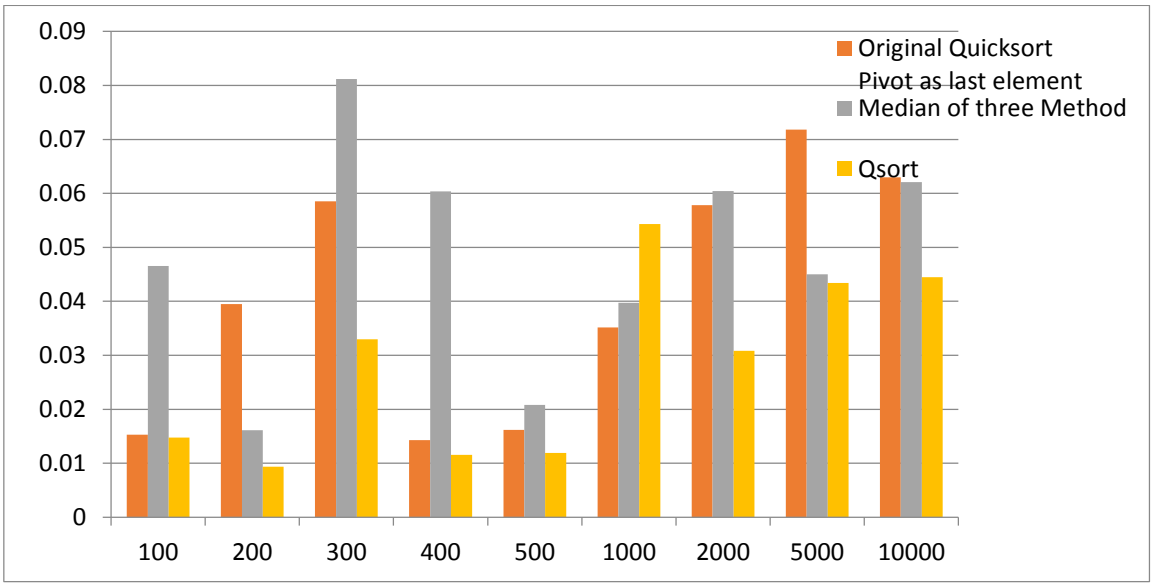

Fig. 2: The Comparison Chart for three algorithms requited to execute unsorted arrays of varying sizes

Table 2: Assessment of time (in seconds) for three algorithms requited to execute sorted arrays with varying sizes

No. of elements in the array(sorted)	Original Quick Sort (Pivot as the last element)	Median of three Method	QSort
100	0.02708	0.04533	0.02032
500	0.03255	0.03658	0.02848
1000	0.4775	0.04698	0.02951
5000	0.2133	0.06411	0.04136
10000	0.6298	0.08402	0.09452

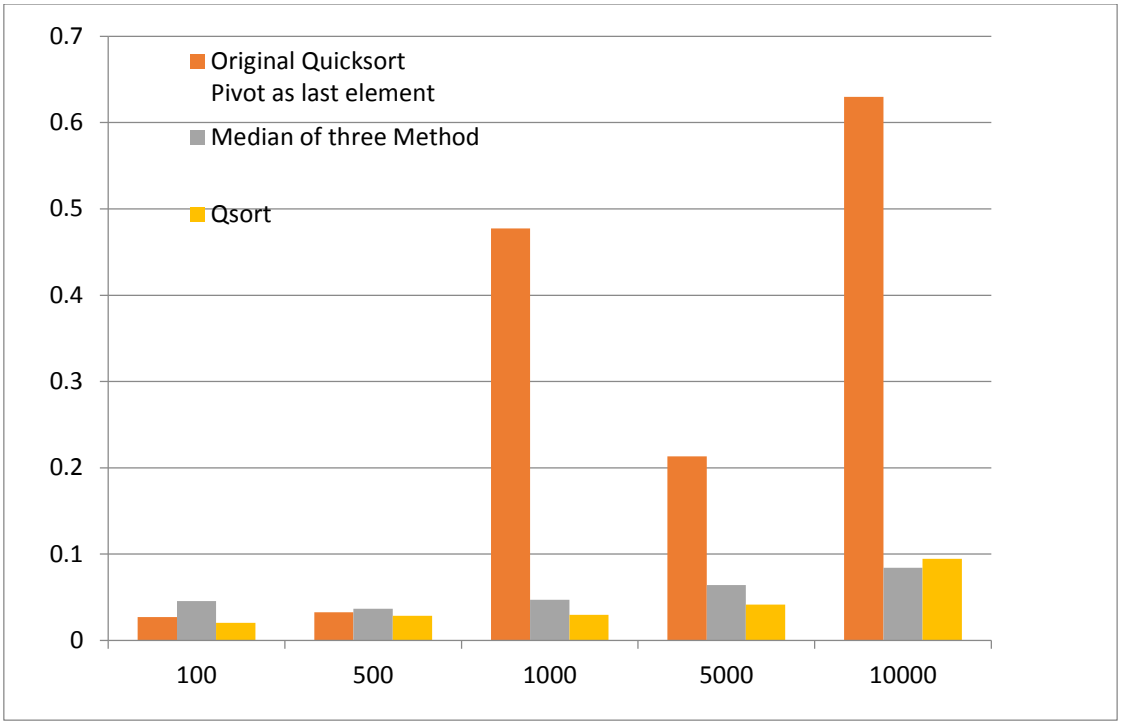

Fig. 3: The Comparison Chart for three algorithms required to execute sorted arrays with varying sizes

6. CONCLUSION

This paper proposed an Advanced Quick Sort (QSort) based on selecting pivot dynamically.
In a recursive call for the right sub-array, the integer average of the values larger than the pivot is passed as the pivot value o determine the pivots for the subsequent level recursive calls. In the same way, in the recursive call for the left sub-array, the integer average of the values less than the pivot is passed as the pivot value to determine the pivots for the subsequent level recursive calls. This technique of picking pivot assists in dividing the array into almost equal halves; thus, improve the performance of the Quick Sort algorithm. The worst case scenario is turned into a best case for the QSort algorithm with **Θ** (n) runtime requirement using the technique of selecting pivot dynamically. The empirical analysis on sorted data confirms that the proposed algorithm runs with **Θ** (n log n) in the worst case.

7. REFERENCES

[1] Hoare, C. A. R. (1961). "Partition: Algorithm 63, Quick Sort: Algorithm 64, and Find: Algorithm 65". Comm. ACM. 4(7), 321-322
[2] Mohammed, A. and Othman M. (2004). "A new pivot selection scheme for Quick Sort algorithm". Suranaree. J. Sci. Technol., 11, 211-215
[3] Mohammed, A. and Othman M. (2007). "Comparative analysis of some pivot selection schemes for Quick Sort algorithm". Inform. Technol. J., 6, 424-427
[4] Van Emden M. H. (1970). "Algorithms 402: Increasing the efficiency of Quick Sort". Communications of the ACM, 563-567
[5] Knuth D.E. (2005). "The Art of Computer Programming. Vol. 3: Sorting and Searching", Addison-Wesley, Reading, Mass.
[6] Dean C. (2006). "A Simple Expected Running Time Analysis for Randomized Divide and Conquer Algorithms". Computer Journal of Discrete Applied Mathematics, 154(1), 1-5
[7] Ledley R. (1962). "Programming and Utilizing Digital Computers." McGraw Hill. Bell D. (1958). The Principles of Sorting. The Computer Journal, 1(2), 71-77
[8] Box R. and Lacey S. (1991). "A Fast Easy Sort". Computer Journal of Byte Magazine, 16(4), 315-321
[9] Kruse R. and Ryba A. (1999). "Data Structures and Program Design in C+ + ". Prentice Hall
[10] Sedgewick R. (1978)." Implementing Quick Sort programs." Comm. ACM, 21(10), 847-857
[11] Janez B., Aleksander V. and Viljem Z. (2000). "A sorting algorithm on a pc cluster", ACM Symposium on. Applied Computing, 2-19
[12] Hoare R (1962)." Quick Sort." the Computer Journal, 4(1), 10-15
[13] Sedgewick R. (1977). "Quick Sort with Equal Keys". Siam J Comput.,6: 240-287

