Volume 2, No. 2, February 2011
Journal of Global Research in Computer Science
ISSN-2229-371X
RESEARCH PAPER
Available Online at www.jgrcs.info

Engineering of a Quicksort Partitioning Algorithm

D.Abhyankar*[1] and M.Ingle[2]
School of Computer Science & IT
Devi Ahilya University Indore India
deepak.abhyankar@yahoo.co.in[1]
maya_ingle@rediffmail.com[2]

***Abstract*:** One of the most sophisticated sorting algorithm in sorting literature is Quicksort. Though Quicksort has several striking aspects, design of partition function is the central aspect of the Quicksort algorithm. Partitioning is a meticulously researched area in which we find Hoare Partition and Lomuto Partition as two prominent partition algorithms in the literature. Despite the fact that much effort has been targeted on research into partitioning, it seems that partitioning is still inadequately understood and amenable to a right blend of optimizations. Superior partitioning algorithms can be designed using a perfect blend of performance improving measures and a touch of elegance. This paper postulates two novel partition algorithms which are better than the existing ones. Proposed algorithm3 apply some effective optimizations and because of this instruction count gets reduced. Reduced instruction count helps the function in gaining spectacular performance. Presented algorithm4 is an elegant algorithm which is compact and intenselycompetitive from performance point of view.

***Keyword*:** Algorithm, Quicksort, Partition, Lomuto

INTRODUCTION

Quicksort is known to be one of the most efficient sorting algorithms[1]. Though there are several imperative issues in Quicksort but the design of partition function is a cornerstone in the study of quicksort because overall performance depends largely on the performance of a partitioning algorithm. There are several partitioning algorithms in the literature, but almost all Quicksort implementations call either Hoare algorithm or Lomuto algorithm for partitioning [2],[3],[5]. Hoare algorithm seems more economical of time than the Lomuto algorithm because it usually needs lesser number of exchanges. Though basic idea of Hoare algorithm is simple but details are intricate and practical experience suggests that to write a correct implementation of Hoare algorithm is an arduous task. On the other hand Lomuto algorithm ends up in a pellucid program and is a concept that can be implemented with relative ease[4]. Both Lomuto and Hoare algorithm have linear time complexity in worst case means both are asymptotically optimal and differ in performance only by a constant factor. Though these algorithms are asymptotically optimal, development of these partitioning algorithms has not fizzled and these partitioning algorithms can be significantly improved using a perfect blend of performance improving optimizations and a touch of style. This paper argues in favor of two intriguing partition algorithms that are sharpened than the existing ones. Section 2 of the paper focuses on the formal and informal description of Hoare and Lomuto algorithms. Following it is section 3, which elucidates two new partition algorithms inspired from Hoare and Lomuto algorithms respectively. The Result statistics of existing and proposed partition algorithms on some crucially important parameters is analyzed in section 4. Finally, we conclude the paper in section 4. Although the psuedocode illustration of an algorithm and its analysis is stimulating and challenging to the academic mathematician's brain, it seems downright dishonesty from an engineering angle. We have therefore strictly adhered to the rule of presenting the functions in a C++ language in which they can actually be executed on a real machine[8].

EXISTING ALGORITHMS

As stated earlier, literature is imbued with partitioning algorithms, but Hoare and Lomuto Partitioning Algorithms emerge as outstanding algorithms, so the current section describes above mentioned algorithms. Section 2.1 and Section 2.2 are devoted to the informal and formal descriptions of Hoare Partitioning algorithm and Lomuto Partitioning algorithm.

Hoare partitioning algorithm

Hoare partitioning algorithm, published by Sir Charles Antony Richard Hoare in 1962. It is an efficient algorithm to fix the pivot element in the correct position and to partition around the pivot element so that elements larger than the pivot will be fixed on the right side of the pivot and elements lesser than or equal to pivot will be fixed on the left side of the pivot. In addition to that partition function returns the correct position of the pivot. The C++ function below with signature HoarePartition(int *a, int p, int r), where a, p and r represents

array, first location, and last location respectively, performs the partitioning operation according to Hoare Partitioning Algorithm and function swap(int *, int *) is called to exchange values in two variables[4].

Algorithm 1: {Hoare-Algorithm}

```
inline void swap(int& x, int& y)
{
        int temp = x;
        x = y;
        y = temp;
}
int HoarePartition(int* a, int p, int r)
{
        if(p>=r) return -1;  // trivial return for empty array
        int x = a[p];        //   x stores the pivot element
        int i = p;
        int j = r+1;
        while(1)
        {
                do{
                        i++;
                }while((i<=r)&&(a[i]<x));   // searches the
        element larger than
                                        // pivot from Left portion

                do{
                        j--;
                }while(a[j]>x);    //  searches the element
smaller than pivot
          // from the Right portion
                if(i>j) break;
                swap(a[i],a[j]);
        }
        swap(a[p],a[j]);   // swaps the larger element from the
        left with the                           // smaller
        element from the right
        return j;               // returns the location of the pivot

}
```

Complexity& Adaptiveness Issues

Hoare partitioning algorithm like any other partitioning algorithm is invariably of θ(n) time complexity. Hoare partitioning algorithm is an inplace partitioning algorithm means it does not entail extra space to partition the array. It is evident that space complexity is invariably θ(n). An adaptive sorting algorithm can take the advantage of already existing order in an array. It was observed that the swap count is the function of the number of inversions that are present among elements of left partition and elements of right partition in the array. In other words if array is partially sorted or roughly sorted Hoare partitioning algorithm will incur lesser number of exchanges. Unfortunately comparison count is n-1 and do not depend on the number of inversions [7]. So it is plain that swap count of Hoare partition is adaptive but comparison count is not influenced by the already existing order of an array.

Lomuto Partitioning Algorithm

Lomuto partitioning algorithm is a lucid partitioning algorithm. Function LomutoPartition(int *a, int p, int r), where a, p and r represents subarray, first location and last location of subarray, implements the Lomuto partitioning algorithm to find the correct location of the pivot element[6].

Algorithm 2 {Lomuto Algorithm}

```
int LomutoPartition(int *a, int p, int r)
{
        int x=a[r];  // x stores the pivot element
        int i=p-1;
        int j=p;  // j is  loop control variable
        while(j<=(r-1))
        {
                if(a[j]<=x)          // a[p] to a[r-1] elements
will be compared with
        // pivot
                {
                   i++;
                   swap(&a[i],&a[j]);
                }
                j++;
        }
                swap(&a[i+1],&a[r]);
                return i+1;  // return the location of the pivot
}

void swap(int *a, int *b)
{
        int temp;
        temp=*a;
        *a=*b;
        *b=temp;
}
```

Complexity&Adaptiveness Issues

Lomuto algorithm like any alternative partitioning algorithm is invariably of θ(n) time complexity. Lomuto partitioning algorithm is an in situ(inplace) partitioning algorithm means it does not necessitate extra space to partition the array. It is apparent that space complexity is always θ(n). An adaptive sorting algorithm can take the advantage of already existing order in an array. Lomuto algorithm lacks the adaptiveness outright. It was found that the total number of exchanges(swap count) depend on the choice of pivot and are not determined by the number of inversions. Unluckily comparison count is (n-1) and is not determined by the inversion count.

PROPOSED MODIFIED ALGORITHMS

The existing partitioning algorithms execute accidental index arithmetic operations, index comparison operations and swaps. This section presents two new algorithms which strive to slash overhead operations. Subsection 3.1 focuses on modified Hoare algorithm which is as efficient as Hoare Algorithm but plucks out redundant instructions from Hoare Algorithm. Section 3.2 reports an algorithm , more elegant than Lomuto algorithm and does cut down the superfluous index manipulation operations executed by original Lomuto Algorithm.

Modified-Hoare Algorithm

The modified Hoare algorithm applies sentinels to cover left as well as right extremes of the array whereas existing Hoare algorithm exerts sentinel at one extreme only and this absence of sentinel at other end severely undercuts the performance of original algorithm. Ingenious use of sentinels reduce the index manipulation operations to optimum level. Swap avoidance is another advantage of this algorithm because it sloughs off 3 instruction swap code. The C++ function ModifiedHoare(int *a, int p, int r), where a, p and r are array, starting index and ending index respectively, articulates new algorithm.

Algorithm 3: {ModifiedHoare-Algorithm}

```
int ModifiedHoare(int* a, int p, int r)
{
        if(a[p]>a[r])
                swap(&(a[p]),&(a[r]));  // Sentinel at both
ends
        int x = a[p];   //  x stores the pivot and location p is
vacant now.
        while(1)
        {
                do{
                        r--;
                }while(a[r]>x);   // search the smaller
element in right
          // portion.
                a[p]=a[r];              // location r is vacant
now.
                do{
                        p++;
                }while(a[p]<x);  // search the larger element
in left portion.
                if(p<r)
                        a[r]=a[p];   // location p is vacant
now.
                else{
                        if(a[r+1]<=x)
                                r++;
                        a[r]=x;
                        return r;   //  return the location of
the pivot
                }
        }
}
```

Complexity and Adaptiveness Issues

Modified Hoare algorithm like any other alternative partitioning algorithm is invariably of $\theta(n)$ time complexity. Modified Hoare partitioning algorithm is an *in situ* partitioning algorithm means it does not occupy extra space to partition the array. It is easy to observe that space complexity is invariably $\theta(n)$. An adaptive sorting algorithm can take the advantage of already existing order in an array. Like Hoare algorithm comparison count of modified Hoare is not affected by existing order in the array but swap count solely depends on the existing order in the input. On the partially ordered input algorithm will incur lesser number of exchanges.

Modified Lomuto Algorithm

This section illustrates modified Lomuto algorithm which is more refined than Lomuto Algorithm. Lomuto algorithm is terse and elegant, however the administrative overhead for the index manipulation and swaps is relatively high. It would seem particularly desirable to slash the administrative overhead operations. This, however, can easily be remedied by Modified Lomuto algorithm that casts aside superfluous index manipulation and swap operations. Modified Lomuto algorithm exhibits that elusive but essential ingredient known as style and touch of elegance. This partitioning algorithm is now formulated in the form of a c++ function. The C++ function ModifiedLomuto(int *a, int p, int r), where a, p and r represent subarray, first location and last location respectively, asserts the new partitioning algorithm.

Algorithm 4: {ModifiedLomuto-Algorithm}

```
int ModifiedLomuto(int* a, int p, int r)
{
        int x = a[p];  // x stores the pivot element
        int i = p; // location i is vacant
        int j = r;
        while(1)
        {
                while(a[j]>x)
                        j--;
                if(j<=i)
                        break;  // terminates the outer loop
                a[i]=a[j];
                a[j]=a[i+1];
                i++;
        }
        a[i]=x;
        return i;  // returns the location of the pivot
}
```

Complexity and Adaptiveness Issues

Modified Lomuto algorithm like any alternative partitioning algorithm is invariably of $\theta(n)$ time complexity in every case. Modified Lomuto partitioning algorithm is an inplace partitioning algorithm means it does not occupy extra space to partition the array. It is effortless to detect that space

complexity is θ(n) without exception. An adaptive sorting algorithm can take the benefit of already existing order in an array. It was noticed that like Lomuto algorithm Modified Lomuto algorithm flatly lacks the adaptiveness.

RESULTS AND DISCUSSION

This section is devoted to the comparative study of results of existing and proposed algorithms on some important parameters. on the one hand Modified Hoare algorithm applies sentinel values at both array extremes to control the loops and thus cuts down the number of index comparisons and on the other hand original Hoare algorithm applies sentinel at one extreme only and suffers a fairly high index manipulation overhead. Table 1 contrasts Hoare Algorithm (Algorithm 1) with Modified Hoare Algorithm (Algorithm 3) on the basis of index comparisons for different input sizes.

Table 1 (On Index Comparisons)

	Algorithm 1	**Algorithm 3**
N	**Best and Worst case**	**Best & Worst case**
100	103	1
200	203	1
1000	1003	1
1200	1203	1
1400	1403	1
2000	2003	1

Modified Hoare Algorithm (Algorithm 3) drastically curtails the number of instructions required to swap two values. Table 2 below contrasts Hoare Algorithm with Modified Hoare Algorithm on the basis of number of instructions executed for exchanging two values.

Table 2 (On swap operations)

	No. of operations for interchanging values.	
N	**Algorithm 1**	**Algorithm 3**
100	150	100
200	300	200
1000	1500	1000
1200	1800	1200
1400	2100	1400
2000	3000	2000

Table 3 contrasts the Lomuto algorithm(Algorithm 2) with modified Lomuto algorithm(Algorithm 4) on frequency of index arithmetic operations. It was perceived that modified Lomuto algorithm is more economical of time than Lomuto algorithm in terms of index manipulation and other operations.

Table 3 (On Index Arithmetic operations)

	Algorithm 2			Algorithm4
N	**Best case**	**Average**	**Worst**	**Always**
100	103	150	202	99
200	203	300	402	199
1000	1003	1500	2002	999
1200	1203	1800	2402	1199
1400	1403	2100	2802	1399
2000	2003	3000	4002	1999

It can be discerned from the comparisons made above that the proposed algorithms perform lesser administrative overhead operations than, required by existing partitioning algorithms. The graphs below show the results obtained by comparing existing partitioning algorithms with proposed partitioning algorithms. Modified Hoare Algorithm employs sentinel values to control loop iterations, which drastically curbs the number of index comparisons. The index comparisons are thus minimized in modified Hoare Algorithm.

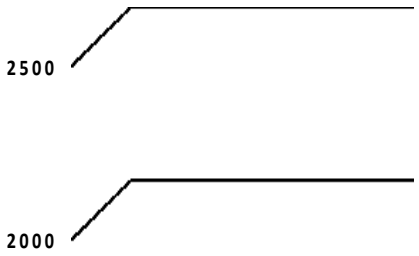

Figure. 4 Shows number of index comparisons along y-axis and input size along x-axis.

Hoare algorithm executes a three instruction *swap* operations to interchange two values and modified Hoare Algorithm competently manages the same in two instructions. Modified Hoare Algorithm thus needs lesser number of instructions to interchange. The number of instructions modified Hoare Algorithm requires to swap is just 2/3rd of instructions executed by Hoare Algorithm. The graph displays the results.

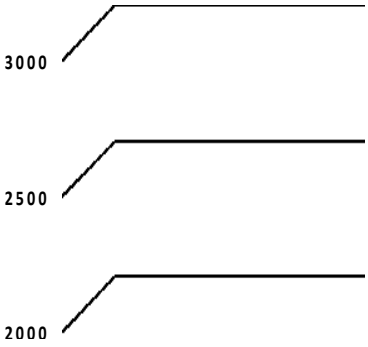

Figure.5Shows number of swaps

Modified Lomuto Algorithm substantially reduces the number of index manipulation operations, on average compared to Lomuto Algorithm. The difference is though negligible in best case of Lomuto Algorithm but is significant in average and worst case of Lomuto Algorithm. Graphs below present average & worst cases.

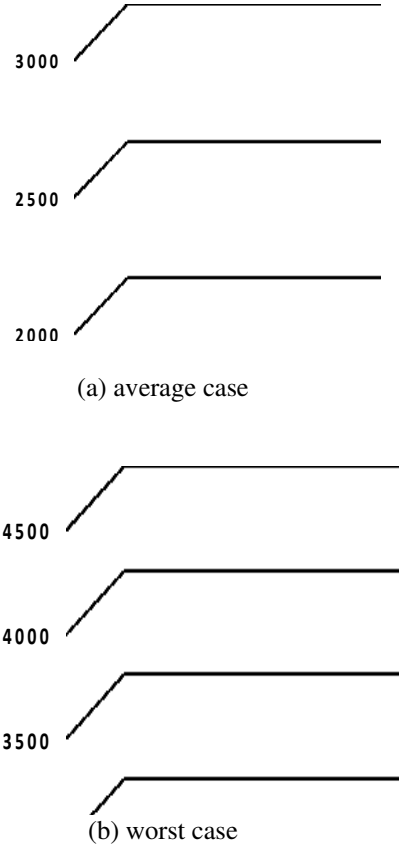

Figure.6 (a & b) shows number of index arithmetic operations

Lomuto partitioning algorithm and Hoare partitioning algorithm have ample scope of improvement. Presence of excessive index manipulation operations and swap operations impede the algorithm's speed. The algorithms presented in this paper address these problems and effectively reduce the instruction count to gain the speed. Suggested algorithms effectively lower the index manipulation overhead to bare minimum. Modified Lomuto algorithm seems to be more elegant than the Lomuto algorithm because of its compact code. Above tables and graphs are helpful to have a rough comparison , however these graphs and tables do not take into account the efforts expended on cache miss, page faults and branch mispredictions. For practical purposes, however it is better to have time profiling of functions to shed light on algorithms under study.

This research adapted a pragmatic approach to conduct a comparative study of existing algorithms and presented algorithms. Netbeans 6.7 was installed for profiling and was highly instrumental in preparing reliable statistics. We have generated random input for our empirical study. Table 4 contrasts the time profiling of Hoare algorithm with proposed algorithm. Figure 4.4 contrasts the same graphically. Table 5 compares the time profiling of Lomuto algorithm with proposed algorithm. Figure 4.5 compares the same graphically. Empirical comparative study revealed that proposed algorithms present drastic improvement over conventional partitioning algorithms. It is important to note that cache miss, page faults and branch miss predictions affect the time spent by a program; hence no algorithm is emerging as a clear winner in terms of time taken.

To conclude this set of partitioning methods, we shall try to contrast their effectiveness. Performance of Hoare partition is not bad, but the performance improvement of Modified Hoare over Hoare partition is spectacular. Unfortunately both Hoare as well as Modified Hoare algorithms show a definite dislike for lucidity and elegance. They are not the most compact either, still Modified Hoare is more compact than Hoare algorithm. If we choose an algorithms from a practical angle then modified Hoare is an ultimate partitioning algorithm and Lomuto algorithm is definitely the worst among all compared. Still from an aesthetic point of view Lomuto algorithm is lucid, elegant and terse but the improvement of Modified Lomuto algorithm over Lomuto algorithm is tangible. Even from performance point of view Modified Lomuto algorithm is better than the Lomuto algorithm.

Table 4 (Time profiling)

	Hoare(Time in ms)	Proposed(Time in ms)
N	**Random Case**	**Random Case**
10000	12.7	.724
20000	5.31	1.93
30000	15.3	2.30
40000	19.7	5.60
50000	11.2	19.9
60000	23.2	2.68
70000	15.4	2.4
80000	3.25	5.58
90000	15.7	14.3
100000	38.2	10.8

Table 5(Time profiling)

	Lomuto(Time in ms)	Proposed(Time in ms)
N	**Random Case**	**Random Case**
10000	2.1	.656
20000	6.5	3.59
30000	5.45	4.61
40000	4.36	2.80
50000	19.6	10.7
60000	18.9	7.32
70000	26	2.17
80000	19.2	2.49
90000	22.2	11.0
100000	25.4	3.18

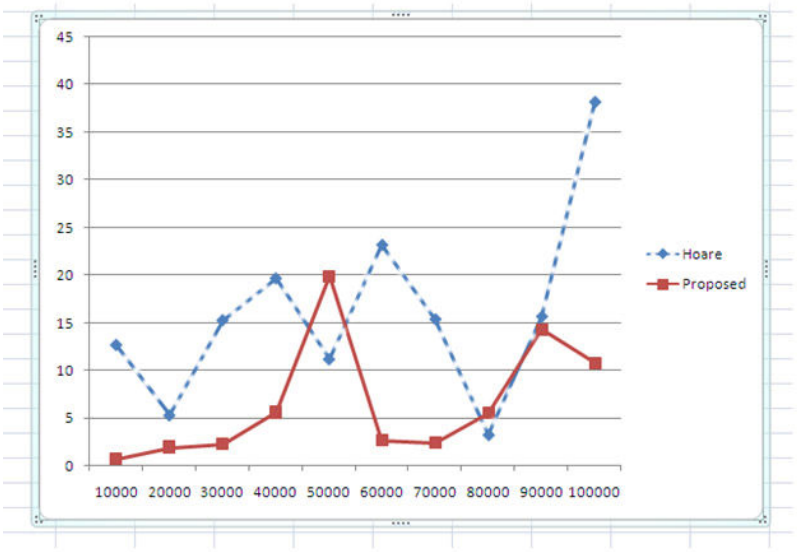

Figure.7(Time profiling)

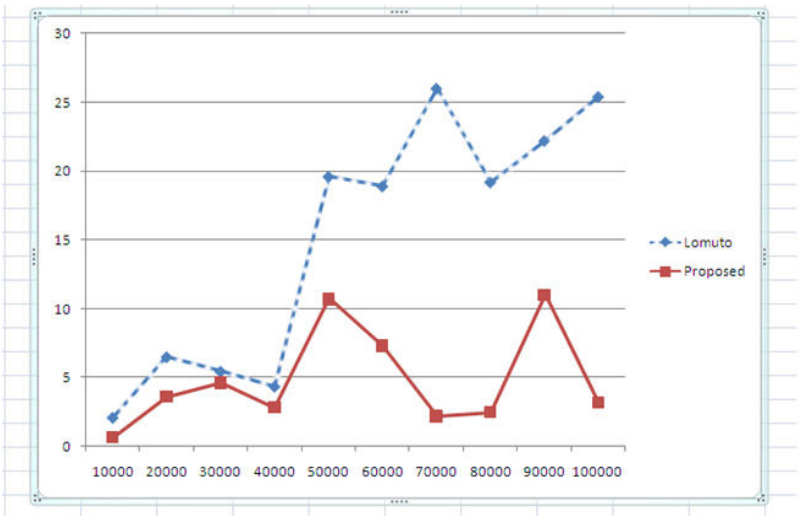

Figure.8 (Time profiling)

REFERENCES

[1] D. E. Knuth, The Art of Computer Programming, Vol. 3, Pearson Education, 1998.

[2] C. A. R. Hoare, "Quicksort," Computer Journal5(1) , 1962, pp. 10-15.

[3] S. Baase and A. Gelder, Computer Algorithms:Introduction to Design and Analysis, Addison-Wesley, 2000.

[4] J. L. Bentley, "Programming Pearls: how to sort," Communications of the ACM, Vol. Issue 4, 1986, pp. 287-ff.

[5] R. Sedgewick, "Implementing quicksort Programs," Communications of the ACM, Vol. 21, Issue10, 1978, pp. 847-857.

[6]T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001.

[7] G. S. Brodal, R. Fagerberg and G. Moruz, "On the adaptiveness of Quicksort," Journal of Experimental AlgorithmsACM, Vol. 12, Article 3.2, 2008.

[8] N. Wirth, Algorithms and Data Structures, © N. Wirth 1985 (Oberon version: August 2004)