WULFENIA JOURNAL KLAGENFURT, AUSTRIA ISSN:1561-882X

Vol 19, No. 10;Oct 2012

Enhancing QuickSort Algorithm using a Dynamic Pivot Selection Technique

Abdel latif Abu Dalhoum[1*] (Corresponding author), Thaer Kobbay[1], Azzam Sleit[1*],Manuel Alfonseca[2],Alfonso Ortega[2]

[(1)]King Abdullah II School for Information Technology, University of Jordan

PO 13898, Amman 11942, Jordan

Tel: 962-797628000 E-mail: a.latif@ju.edu.jo

[(2)]E.T.S. Informática, Universidad Autónoma de Madrid, 28049 Madrid, Spain

[*] On sabbatical leave from University of Jordan

Abstract

Sorting is one of the most researched problems in the field of computer science, where several algorithms have been proposed. For large input size, QuickSort has proved to be the fastest sorting algorithm. This paper presents an intelligent QuickSort algorithm based on a dynamic pivot selection technique to enhance the average case and eliminate the worst case behaviors of the algorithm. The suggested dynamic pivot selection technique is data-dependent to increase the chances of splitting the array or list into relatively equal sizes in order to reduce the number of recursive calls made for the Quicksort algorithm. Furthermore, the modified algorithm converts the worst case into a best case behavior with Θ(n) execution time. The algorithm is intelligent enough to recognize a sorted array or sub-array that doesn't require further processing.

Keywords: QuickSort, MQuickSort, Dynamic Pivot Selection, Divide & Conquer, Median-of-Three Rule, Median-of-Five Rule**.**

1. Introduction

The sorting problem is one of the most researched problems in the field of Computer Science since many other algorithms require sorted lists as a prerequisite to reduce their execution time and enhance performance. The QuickSort algorithm (Hoare, C. A. R., 1961) (Hoare R., 1962) is considered to be the fastest sorting algorithms based on different studies (Sedgewick R., 1977) (Van Emden M. H., 1970) (Knuth D.E., 2005). Quicksort follows the technique of divide and conquer by recursively splitting each array into two sub arrays, which makes it easier to solve smaller problems than a single larger one (Dean C., 2006) (Ledley R., 1962). In Quicksort, a pivot is selected from the unsorted array and used to split the array into two sub arrays for which the same algorithm is called recursively until the sub arrays have size one or zero. For an input size n, the QuickSort algorithm has an average runtime complexity of $\Theta(n \log n)$ and a worst case scenario of $\Theta(n^2)$ when processing an already sorted list while picking the largest element as a pivot.

The runtime of the Quicksort algorithm mainly 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 runtime will be:

$T(n) = n + T(n-c)$, where c is a constant. Thereby,

$$T(n) = \Theta(n^2) \qquad (1)$$

However, if splitting constantly results almost equal size subarrays, the runtime complexity of Quicksort will be:

$$T(n) = n + T(n/2).$$

Thereby,

$$T(n) = \Theta(n \log n) \qquad (2)$$

Splitting the array into almost equal halves guarantees the best performance for the QuickSort algorithm and reduces the number of recursive calls which will eventually reduce execution time.

Two main enhancements can be applied to the basic QuickSort algorithm. The first modification suggests stopping the recursive calling to QuickSort when the size of the input becomes relatively small and only few elements are out of order. In such case, it would be better to use Insertion sort which performs in linear time for almost ordered sub arrays (Bell D., 1958) (Box R. and Lacey S., 1991) (Kruse R. and Ryba A., 1999). The second enhancement is concerned with the pivot selection technique which has proven to be the most decisive factor in dividing the array into sub arrays. Several techniques have been proposed in order to avoid the worst case scenario previously explained in (1). The original QuickSort algorithm uses the left-most or the right-most element as a pivot which can easily cause the worst case behavior when sorting a sorted or almost list. Another technique is the random pivot selection technique which reduces the probability of occurrence for the worst case scenario. The Median-of-Three splitting technique (Sedgewick R., 1978) suggests picking the median of the values stored in the first, last and ((first + last) /2) indexes as a pivot. This technique reduces the chances of the worst-case scenario and increases the chances of the average case behavior of the algorithm. However, it does not guarantee splitting the array into equal halves and may bring about the worst case behavior of QuickSort although it's unlikely to happen. Figure 1(a) shows the splitting of an eleven-element sample array. Switching the values of the fifth and sixth elements of the array will result in a more balanced splitting as shown in Figure 1(b). It is clear that the performance of the algorithm is sensitive to slight modifications for the contents of the array.

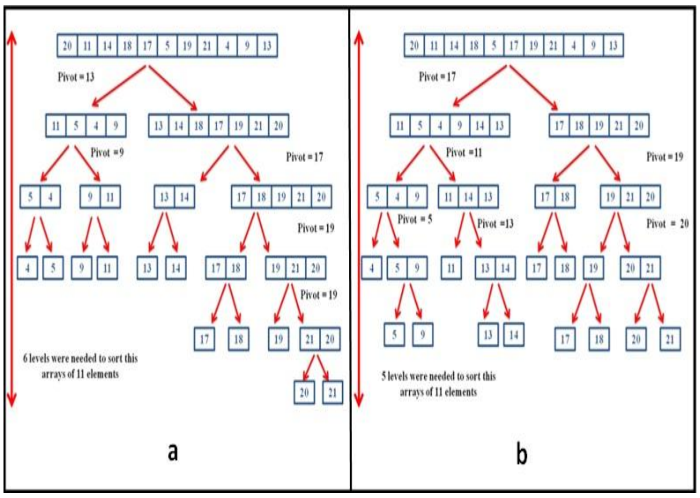

Pivot = 13
Pivot =9
Pivot = 17
Pivot = 19
Pivot = 19
6 levels were needed to sort this arrays of 11 elements
a
Pivot = 17
Pivot =11
Pivot = 19
Pivot = 5
Pivot =13
Pivot = 20
5 levels were needed to sort this arrays of 11 elements
b

Figure 1. (a) Applying the QuickSort algorithm on an eleven-element array using the Median-of-Three splitting technique. (b) Applying the algorithm on a slightly reordered array with the same contents and size.

The Median-of-Five with random index selection technique (Janez B. et al., 2000) is a modification for the previous technique by adding the values stored in two randomly picked indexes to the values stored in the first, last and ((first + last) /2) indexes. Although, this technique may provide a more balanced splitting than the previous ones, it can still suffer from the same setbacks of the previous techniques. Another modification for the previous method which aims to reduce the overhead associated with the random number generation picks the median of the five values stored in fixed indexes as the pivot; namely, first, ((first + last) /4), ((first + last) /2), (3 *(first + last) /4) and last (Mohammed, A. et al., 2004). Although, this technique reduces the overhead of the random number generation by using five fixed indexes, it still requires time to pick the median of five elements at each recursive call.

Other techniques involving the Median-of-Seven and Median-of-Nine either with or without random index selection were proposed in (Mohammed, A. et al., 2004). Although, increasing the number of elements may provide a more balanced split, the time needed to pick the pivot at each recursive call increases as the number of elements increases.

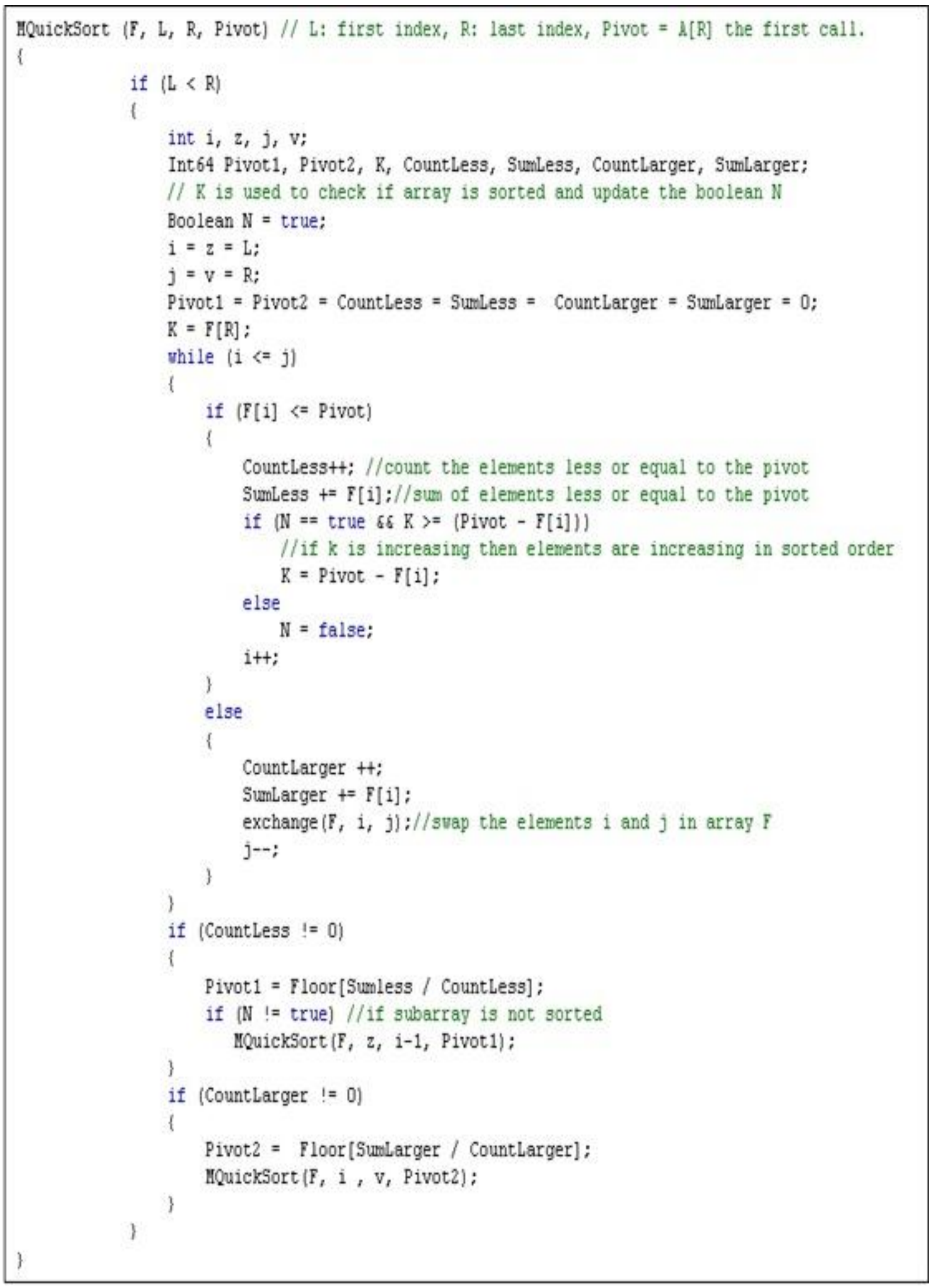
The main drawback of the previous techniques is that the selection of the pivot is based on a specific number of elements which does not necessarily reflect the nature of the array. When using any of these pivot selection techniques, the worst case behavior of the QuickSort algorithm may still occur.

This paper is organized as follows. Section 2 presents the modified QuickSort (MQuickSort) algorithm using the dynamic pivot selection technique. Sections 3 and 4 discuss behavioral analysis of MQuickSort in comparison with QuickSort algorithm using various existing pivot selection techniques. This paper concludes with section 5.

2. Dynamic Pivot Selection Technique

This section proposes a pivot-selection technique based on the values of every element in the array to split the array into relatively equal halves for each recursive call which in turn reduces the number of recursive calls and the overall execution time of QuickSort. The proposed technique aims to ensure equal splitting for the array. Therefore, it drives the worst case behavior to be O(n log n). The proposed technique also verifies an already sorted array or sub array which is done while comparing the elements of the array to the pivot. If the array is already sorted, it will not be processed any further which transforms the $O(n^2)$ complexity into the best case behavior of the algorithm; i.e. O(n). The proposed technique operates as follows; at first, the pivot value is chosen to be the value of the rightmost element of the array. Each element value will be compared with the pivot value and two counters 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; namely, CountLess and CountLarger, respectively. The sum of the values of the elements smaller than the pivot and the sum of those larger than the pivot are stored in SumLess and SumLarger, respectively. These variables are then used to calculate the next pivots for the recursive calls. The integer average of the values smaller than the pivot is passed as the pivot value of the recursive call for the left sub array. Likewise, the integer average of the values larger than the pivot is passed as the pivot value of the recursive call for the right sub array. This pivot selection technique helps in successively splitting the array into nearly equal halves which in turn improves the efficiency of the QuickSort algorithm. A Boolean variable is utilized by the algorithm to recognize an already sorted array or sub array which reduces the number of recursive calls. Along with the reduction in recursive calls, the proposed technique converts the worst-case scenario for the classical QuickSort algorithm into a best case scenario with $\Theta(n)$ runtime. The modified algorithm MQuickSort is provided in Figure 2.

The dynamic pivot selection technique does not depend on the position of the values stored in the array. Figure 3 displays the splitting of the arrays in Figure 1(a) and (b) where the same splitting tree is generated for both arrays which indicate that the MQuickSort algorithm is not sensitive to the order of elements in the array.



```
MQuickSort (F, L, R, Pivot) // L: first index, R: last index, Pivot = A[R] the first call.
{
            if (L < R)
            {
                int i, z, j, v;
                Int64 Pivot1, Pivot2, K, CountLess, SumLess, CountLarger, SumLarger;
                // K is used to check if array is sorted and update the boolean N
                Boolean N = true;
                i = z = L;
                j = v = R;
                Pivot1 = Pivot2 = CountLess = SumLess =  CountLarger = SumLarger = 0;
                K = F[R];
                while (i <= j)
                {
                    if (F[i] <= Pivot)
                    {
                        CountLess++; //count the elements less or equal to the pivot
                        SumLess += F[i];//sum of elements less or equal to the pivot
                        if (N == true && K >= (Pivot - F[i]))
                            //if k is increasing then elements are increasing in sorted order
                            K = Pivot - F[i];
                        else
                            N = false;
                        i++;
                    }
                    else
                    {
                        CountLarger ++;
                        SumLarger += F[i];
                        exchange(F, i, j);//swap the elements i and j in array F
                        j--;
                    }
                }
                if (CountLess != 0)
                {
                    Pivot1 = Floor[Sumless / CountLess];
                    if (N != true) //if subarray is not sorted
                       MQuickSort(F, z, i-1, Pivot1);
                }
                if (CountLarger != 0)
                {
                    Pivot2 =  Floor[SumLarger / CountLarger];
                    MQuickSort(F, i , v, Pivot2);
                }
            }
}
```

Figure 2: The MQuickSort Algorithm

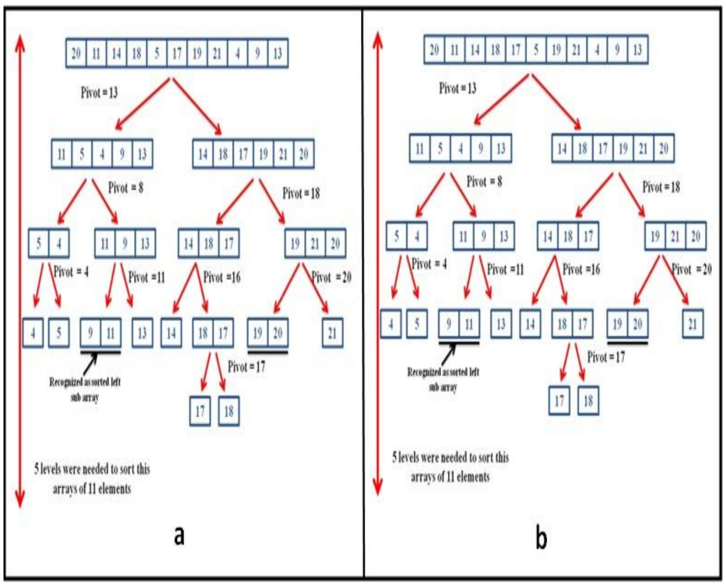

Pivot = 13
Pivot = 8
Pivot = 18
Pivot = 4
Pivot = 11
Pivot = 16
Pivot = 20
Pivot = 17
Recognized as sorted left sub array
5 levels were needed to sort this arrays of 11 elements
a
b

Figure 3. Applying the MQuickSort algorithm with the dynamic pivot selection technique on the arrays of Figure 1(a) and 1(b) results with identical splitting trees.

The MQuickSort algorithm may get affected in case of the existence of some very small and/or large elements compared to the other elements in the array. The example shown in Figure 4 suggests that the MQuicksort algorithm still performs better than the Median-of-Three rule based QuickSort algorithm in terms of the height of the splitting tree. Both algorithms require the same number of recursive calls.

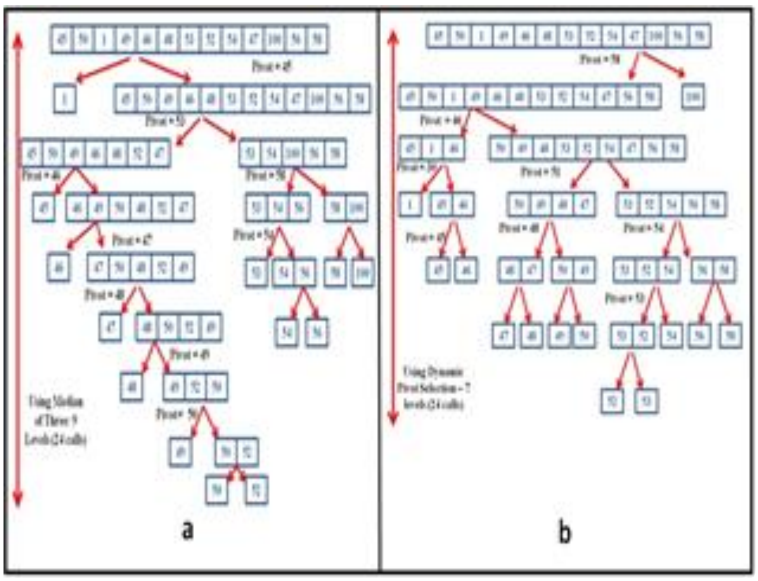


Figure 4. (a) Applying the QuickSort algorithm using the Median-of-Three splitting technique on an eleven element array with extreme values. (b) Applying the MQuickSort algorithm on the same array of (a).

3. Performance Analysis

The MQuickSort algorithm initially selects the contents of the rightmost component of the array as a pivot. For any iteration and while going over the contents of the array to calculate the mean of the elements less than the pivot and that of the elements larger than the pivot, it utilizes a Boolean variable which will have a TRUE value if the array is sorted in order to stop recursive calls. Consequently, only one recursive call is required for a sorted array with $\Theta(n)$ runtime requirement for the first and only required iteration. The above is also true in case of sorted sub-arrays which helps in reducing the execution time.

For an average case with the contents of the array not having extreme values, the MQuickSort algorithm tends to calculate the required pivots for the next two recursive calls during the current iteration in order to split the array into approximately two equal sub-arrays. The runtime requirements of the average case can be expressed as $T(n) = n + T((n/2)+c) + T((n/2)-c)$, where c is a constant. The n component of T(n) is the time needed to calculate the pivot for the next iteration while $T((n/2)+c)$ and $T((n/2)-c)$ are the times needed for recursively sorting the two approximately equal sub-arrays. Solving the recurrence results with $T(n) = \Theta(n \log n)$ time requirement for the average case behavior of the MQuickSort algorithm.

The worst case scenario of the MQuickSort algorithm takes place when there are just a few elements of the array with extreme values. The MQuickSort algorithm tends to isolate the extreme values in early iterations and then proceed with the remaining elements with recursive calls which will strictly have an average case behavior with $\Theta(n \log n)$ runtime requirement.

4. Experimental Analysis

In order to experimentally evaluate the performance of the MQuickSort algorithm we wrote C# implementation for MQuickSort and four versions of QuickSort using pivots as last element, Median-of-Three, Median-of-Five and Median-of-Five with fixed index. For each run of the experiment, we randomly generated the contents of fifty arrays to be sorted using the above-mentioned five algorithms on an Intel core2 duo 2.00 GHz T6400 CPU with 3 GB RAM computer. The average execution time of all arrays is calculated in milliseconds. Experiments were conducted in three different scenarios.

Scenario1:

This scenario represents sorted arrays with sizes from 100 to 100,000 elements. Table 1 shows the runtime requirements in milliseconds for all five algorithms. It is clear that the MQuickSort algorithm with dynamic pivot selection constantly outperforms QuickSort using all the other pivot selection techniques. Results also show that QuickSort using the Median-of-Three rule performs better than both of the Median-of-Five pivot selection techniques which indicates how much time is consumed to pick a median of five compared to picking a median of three.

Scenario 2:

This scenario represents randomly generated arrays with values chosen from 1 through Array_Size/10. The objective of this scenario is to compare the behavior of the five algorithms with respect to arrays with duplicate contents. Table 2 shows the runtime requirements of the algorithms. The MQuickSort algorithm exhibits a clear superiority especially for array sizes greater than or equal to 2000. The performance of the QuickSort algorithm using the last element as a pivot tends to be comparable to that when using the Median-of-Three technique. Likewise, the QuickSort algorithm performs almost the same using the fixed and non-fixed Median-of-Five techniques.

Scenario 3:

This scenario represents sorting arrays with no duplication. Table 3 clearly demonstrates the superiority of the MQuickSort algorithm using the dynamic pivot selection technique especially for array sizes greater than or equal to 3000 elements.

5. Conclusion

This article proposed a modified QuickSort (MQuickSort) based on a dynamic pivot selection technique. In order to determine the pivots for the next level recursive calls, the integer average of the values larger than the pivot is passed as the pivot value of the recursive call for the right sub array. Likewise, the integer average of the values less than the pivot is passed as the pivot value of the recursive call for the left sub array. This pivot selection technique helps in successively splitting the array into nearly equal halves which in turn improves the efficiency of the QuickSort algorithm. Using the dynamic pivot selection technique, the worst case scenario in a typical QuickSort algorithm is turned into a best case for the MQuickSort algorithm with $\Theta(n)$ runtime requirement. The MQuickSort algorithm runs with $\Theta(n \log n)$ in the worst case which is confirmed by experiments

Table 1 Comparison of execution time (in milliseconds) for five algorithms with respect to sorted arrays with various sizes.

Array Size	QuickSort using last element as pivot	QuickSort using Median-of-Three	QuickSort using Median-of-Five	QuickSort using Median-of-Five – Fixed	MQuickSort
100	0.1154	0.0236	0.119	0.1103	0.0041
200	0.4465	0.0477	0.2165	0.2227	0.0056
300	0.989	0.0718	0.3367	0.3284	0.0076
400	1.7492	0.0985	0.4408	0.4578	0.0092
500	2.7197	0.119	0.584	0.5712	0.0112
1000	10.8355	0.2504	1.1928	1.1584	0.0205
2000	43.3124	0.5276	2.3969	2.3615	0.039
3000	97.5886	0.8432	3.5861	3.6041	0.0564
4000	173.3558	1.1076	4.8318	4.7605	0.0754
5000	269.8456	1.4571	6.0714	6.0216	0.0934
10000	1080.4153	3.1595	12.283	12.1521	0.1852
20000	4358.348	6.4548	25.1587	25.0068	0.368
30000	9704.39	9.6488	37.9272	38.3948	0.5512
40000	17168.788	13.4588	51.2957	50.8656	0.737
50000	26579.325	16.9567	64.346	65.1338	0.9182
100000	108820.68	36.4823	132.6257	127.4971	1.8873

Table 2 Comparison of execution time (in milliseconds) for five algorithms with respect to arrays of various sizes with duplicate values.

Array Size	QuickSort using last element as pivot	QuickSort using Median-of-Three	QuickSort using Median-of-Five	QuickSort using Median-of-Five – Fixed	MQuickSort
100	0.0256	0.0266	0.1121	0.112	0.0284
200	0.0546	0.0567	0.2406	0.2348	0.0574
300	0.0843	0.0906	0.3615	0.3589	0.0881
400	0.114	0.1192	0.4824	0.4712	0.1171
500	0.1492	0.1569	0.6078	0.5992	0.1489
1000	0.313	0.3321	1.2739	1.2585	0.3066
2000	0.7235	0.7248	2.5183	2.5344	0.6138
3000	1.0937	1.1091	3.8453	3.8243	0.8966
4000	1.4774	1.5264	5.1561	5.1133	1.1811
5000	1.9176	1.9609	6.384	6.3567	1.4332
10000	4.6149	4.558	12.1755	12.1056	2.7234

20000	**11.038**	**11.049**	**23.3832**	**23.324**	**5.3994**
30000	**20.3844**	**19.6543**	**36.9559**	**36.6294**	**7.8735**
40000	**31.8965**	**30.4145**	**51.7696**	**51.6329**	**10.535**
50000	**43.4311**	**43.6966**	**68.5958**	**68.8652**	**13.4729**
100000	**142.0017**	**140.5775**	**188.2229**	**185.163**	**26.1101**

Table 3 Comparison of execution time (in milliseconds) for five algorithms with respect to arrays of various sizes with no duplicate values.

Array Size	**QuickSort using last element as pivot**	**QuickSort using Median-of-Three**	**QuickSort using Median-of-Five**	**QuickSort using Median-of-Five – Fixed**	**MQuickSort**
100	**0.0246**	**0.0282**	**0.1206**	**0.1165**	**0.0272**
200	**0.0585**	**0.0615**	**0.2581**	**0.2217**	**0.06**
300	**0.0819**	**0.0883**	**0.3757**	**0.3674**	**0.0845**
400	**0.1152**	**0.1219**	**0.4917**	**0.4809**	**0.1211**
500	**0.1474**	**0.1557**	**0.6405**	**0.6318**	**0.1512**
1000	**0.3325**	**0.3337**	**1.2369**	**1.2297**	**0.3322**
2000	**0.6903**	**0.7247**	**2.6176**	**2.5894**	**0.7006**
3000	**1.0886**	**1.0759**	**3.9957**	**3.8756**	**1.0841**
4000	**1.4802**	**1.4764**	**5.226**	**5.1649**	**1.4729**
5000	**1.9222**	**1.9003**	**6.5764**	**6.4856**	**1.9075**
10000	**4.0962**	**4.0781**	**13.1933**	**13.0936**	**4.0466**
20000	**8.6898**	**8.5713**	**27.7548**	**27.3385**	**8.0406**
30000	**13.6301**	**13.2745**	**41.4112**	**41.3773**	**12.9442**
40000	**18.5857**	**17.6174**	**55.6076**	**55.5537**	**17.1203**
50000	**23.2952**	**22.5985**	**69.4853**	**69.0916**	**22.114**
100000	**52.3214**	**49.4151**	**142.5502**	**141.7248**	**46.852**

References

Hoare, C. A. R. (1961). Partition: Algorithm 63, Quicksort*:* Algorithm 64, and Find: Algorithm 65. *Comm. ACM*. 4(7), *321-322*

Hoare R.(1962). Quicksort. *the Computer Journal*, 4(1), 10-15

Sedgewick R. (1977). QuickSort with Equal Keys. *Siam J Comput*.,6: 240-287

Van Emden M. H. (1970). Algorithms 402: *Increasing* the efficiency of *Quicksort.* Communications of the ACM, 563-567

Knuth D.E. (2005). The Art of Computer Programming. *Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass.

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

Ledley R. (1962). *Programming and Utilizing Digital Computers*. McGraw Hill.

Bell D. (1958). The Principles of Sorting. *The Computer Journal*, *1*(2), 71-77

Box R. and Lacey S. (1991). A Fast Easy Sort. *Computer Journal of Byte Magazine*, 16(4), 315-321

Kruse R. and Ryba A. (1999). *Data Structures and Program Design in C++*. Prentice Hall

Sedgewick R. (1978). Implementing quicksort programs. *Comm. ACM*, 21(10), 847-857

Janez B., Aleksander V. and Viljem Z. (2000). *A* sorting algorithm on a pc cluster, *ACM Symposium* on. *Applied Computing*, *2-19*

Mohammed, A. and Othman M. (2004). A new pivot selection *scheme* for Quicksort algorithm. *Suranaree.* J. Sci. *Technol.*, *11*, *211-215*

Mohammed, A. and Othman M. (2007). Comparative analysis of some pivot selection schemes for Quicksort algorithm*.* *Inform. Technol. J.*, 6, 424-427