International Research Journal of Mathematics, Engineering and IT
Vol. 4, Issue 1, January 2017 **Impact Factor- 5.489**
ISSN: (2349-0322)
© Associated Asia Research Foundation (AARF)
Website: www.aarf.asia Email : editor@aarf.asia , editoraarf@gmail.com

EMPIRICAL COMPARISON BETWEEN CLASSICAL QUICKSORT AND AN ENHANCED TIME OPTIMAL QUICKSORT

Dr. Mirza Abdulla
Computer Science Department
College of Computer Studies
AMA International University, Bahrain.

ABSTRACT

Quicksort was and is still one of the most practical sorting techniques. Unfortunately, it suffers from a worst case asymptotic running time of $O(n^2)$ on a list of n items. However, its worst case running time can be improved to $\theta(n \log n)$ by using the median of medians technique for pivot finding but at the expense of relative deterioration in average case performance. We compare the running time improvement on classical quicksort when the median of medians is only called when there is mounting evidence that it could help improve the run time performance of the pivot finding method used. We study and present evidence that such a technique can be quite practical and yet have optimal worst case running time.

Keywords: Quicksort, pivot, partitioning, average case performance, median.

1. Introduction.

The main task of computers is to store, manipulate and retrieve data. The manipulation of data may require the rearrangement of data to facilitate future manipulations and retrieval operations. This made searching and sorting fundamental problems in computing, and indeed all applications require some form of rearrangement and retrieval of data. The problem of sorting in particular is to rearrange the data in a particular order, usually ascending or descending. A clear example of the importance of sorting is the telephone directory. Imagine that you want to find the number of a particular person, but the names in the directory are random and compare the search time to that of a directory with lexicographically ordered entries.

Sorting as a problem existed long before the first computer was built. Indeed, it is well known that Radix or Hollerith sort was used in the 1890 USA census and dates back to 1887. Other techniques that date back to the mid of the twentieth century include techniques such as Bubble sort.

Sorting techniques are divided into two main categories: Comparison based and non comparison based. For example the aforementioned Radix sort falls into the non-comparison based sorting techniques, which Bubble sort is a comparison based sorting technique. Comparison based sorting techniques can also be divided into ones that sort incrementally or those that use Divide and Conquer. For example, Bubble, Insertion, and Selection Sort fall into the incremental sort techniques, while others such as Heap sort [1], Merge sort [2], and Quick sort [3] use divide and conquer. Incremental techniques suffer from the fact that they are quite inefficient, though variants such as Shell sort [4], and Enhanced Selection sort [5] attain good asymptotic performance, but they are no match for the more efficient sort techniques that use divide and conquer such as Heap sort or Quick sort.

Sorting as a problem is known to require $\Omega(n \log n)$ comparisons in comparison based sorting. Heap sort and Merge sort attain this asymptotic bound and therefore, optimal sorting techniques up to a constant factor. Quick sort on the other hand has a worst case asymptotic performance of $O(n^2)$. however, the average case performance of quick sort is $O(n \log n)$. Nowadays there are hundreds of sorting techniques, but quick sort is still considered the most practical sorting technique.

The sorting idea of quick sort is quite simple, and it is based on a partitioning scheme that would accept a list of items and splits it into two lists based on a pivot. All the items in one list contain data items with keys not more than the pivot, and all the items in the other list contain keys with value greater than the pivot. This way we can arrange the items so that all data items with key values not more than the pivot appear before all data items with key values more than the pivot. By recursing on both lists we eventually obtain a sorted list. The pivot in this sorting technique plays an important role in determining the efficiency of the technique. For example, if we always choose the least or highest key valued element as the pivot, we end up with an inefficient sorting process whose asymptotic running time is $O(n^2)$. However, if we always use the median of the given list as the pivot for the partitioning step, we end up with two lists of more or less the same size. In this case the running time of the sorting process is $O(n \log n)$, which matches the optimal asymptotic bound for comparison based sorting.

In practice it is unlikely to always have the smallest or greatest element as the pivot for the partitioning process. Indeed, even if we meet such a case at some point in the sorting process, it is more likely to get a more even pivot in the next round. However, despite its superior average performance, it still lacks the theoretical "prestige" of having $\theta(n \log n)$ worst case asymptotic bound. One can employ available techniques for finding the median of a list in order to find a suitable pivot which can then be used to partition the lists during the sorting process. However, this increases the constant factor in the average running time of the algorithms which makes it less efficient and not as practical as some of the other available sorting techniques, and certainly less practical than the classical quick sort.

[6] Presents a technique to deal with the problem of making the worst case running time of quick sort be $\theta(n \log n)$ while at the same time leaving the average case running time mostly unaffected by the modifications made. The technique is not to employ the median finding algorithm every time we need to find the pivot, rather, only call a median finding routine under restricted conditions that guarantee a worst case running time of $\theta(n \log n)$, and yet improve the average running time or at least don't substantially affect it. The restriction on the use of the median of medians is through the use of a threshold level which records the number of consecutive recursive calls in quicksort result in unevenly partitioned lists. The degree of unevenness was not defined in [6], but in this paper we use 10 as evenness factor, or in other words if one of the partitioned lists is at least 10 times the size of the smaller list we regards the partitioning as uneven.

In this paper we perform empirical tests to study the effect of the modifications suggested by [6] on the practicality of the quick sort. We study the modified quick sort under various degrees of thinning the original list to extract an "acceptable" pivot as well as the value of the threshold after which we resort to the median function to find a pivot.

In section 2 we present the thinning technique used in our study, and in section 3 we present the quick sort technique given by [6], which we would refer to in this paper as the *Enhanced Quicksort*. Section 4 presents the results of comparison between the running times of classical versus the enhanced quick sort when we use exactly the same data for both techniques.

2. The Thinning algorithm.

[7] Produced the first $O(n)$ time algorithms to find the median or the kth item in a list of n items without resorting to sorting the list. The steps can be summarized as:

1. Partition the list of n elements into $n/5$ groups of 5 elements each.
2. Find the median of each group. In our implementation the median of the group is found using insertion sort.
3. Find the median of medians by recursion on the group medians.
4. If found median of medians is indeed the median of the whole list we can stop with answer
5. Otherwise partition the original list on this median and the larger partition would contain the median of the whole list and all we need to do is to find the item in that list whose rank would make it the median of the original list.

According to [6] the median shouldn't be called on the whole list, but a small fraction of the given list. The fraction of items chosen from the list is taken by inspecting every dth element, for some constant $d \geq 2$. The function for thinning the list is as follows:

```
function ThinList(a[], l, r, d, j)
    i = l;
    while (i+jd  ≤  r){
        m1 = i;
        t = i + jd;
        for(s= i+j; s<t; s+=j)        if (a[s] < a[m1]) m1 = s;          //takes O(n/j) time
        swap(i, m1);                                                     //every item greater than
d items
        i+=jd;
        }
    i = l;
    while (i + jd^2  ≤  r){
        m1 = i;
        t = i + jd^2;
        for(s= i+jd; s<t; s+= jd)   if (a[s] > a[m1]) m1 = s;          //takes O(n/jd) time
```

```
        swap(i, m1);                                    //every item less than d items
        i+= d^2;
        }
    i = jd^2;
    return mom(a[l]; a[l+l]; a[l+2l]; . . . ; a[t])            // t = l + (r-l)/
}                                                   // takes O(n/d^2) time.
```

The ThinList function scans the list during the first while loop and inspects every *d*th element in the list. It finds the minimum of these items for every *d* inspected items and swaps that minimum with the *i*th item in the list. Thus By the end of this step we know that the first item in every group of d^2 elements in the original list is at least less than or equal to *d* elements in that group. The ThinList function inspects the lists again in the second while loop, but then every d^2th element in groups of d^3 elements in the original list. However, this time it finds the maximum of these and swaps it with the first element in the group. This time we are guaranteed that the maximum of each group of d^3items has at least *d* elements in a group of value greater than or equal, and similarly *d* items of value less than or equal.

These n/d^3 elements are moved to the beginning of the list (this step was performed during the while loops to save time) and the median of medians algorithm is applied only to this portion of size n/d^3 of the list.

2. The QuickSort algorithm.

We don't put any restriction on the use of a particular quicksort algorithm in this paper. As a matter of fact any quicksort implementation that uses a constant number of values to determine the pivot will do, even ones that use dual or multiple pivots. We shall refer to such an implementation as *Classical Quicksort*. According to [6] the enhanced quickSort algorithm to be used is exactly the classical quicksort, but with a minor modification. The modification is through the use of the integer variable *lvl* and the integer constant: *threshold*. Quicksort uses the normal pivot evaluation prior to the partitioning step. A check is made prior to recursive call for quicksort on the partitioned lists to see if one of the partitioned lists is greater in size than the other by a factor of say 10 or more then the value

of the variable *lvl* is increased by 1. The increase of 1 to the *lvl* variable is only applied in the call to the larger of the two partitions. When the value of *lvl* reaches a level determined by the threshold constant we set the *lvl* variable again to 0 and call the ThinList function explained earlier to find the pivot instead of the normal pivot finding step. This way the median of medians is called on a small fraction of the list and only when the normal pivot finding technique keeps giving us pivots that produce highly uneven lists for a number of consecutive recursive calls.

```
• if (lvl ≥ threshold){
      lvl=0;
      pivotIndex =ThinList(input, low, high, d, lvl);
      pivot=input[pivotIndex];}
    else pivot = normal pivot finding technique;
• Perform normal partitioning step
• Let sizeLeft be size of the list with elements ≤ the pivot;
  Let sizeRight be size of the other list with elements > the pivot;

• if (sizeLeft > 10∗ sizeRight){lvl++;
      quicksort (input, low, newHigh, d, lvl, threshold);
      quicksort (input, newLow, high, d, 0, threshold);
    }
  elseif (sizeRight > 10∗ sizeLeft){lvl++;
      quicksort (input, low, newHigh, d, 0, threshold);
      quicksort (input, newLow, high, d, lvl, threshold);
    }
  else {
      quicksort (input, low, newHigh, d, 0, threshold);
      quicksort (input, newLow, high, d, 0, threshold);
    }
```

3. Empirical comparison

3.1. *Tests made*

The tests made for comparison between the enhanced quicksort and classical quicksort using Java. The comparisons were made on the same data as input to each routine using: first item, median of three, and a random item, as the pivot in the classical quicksort and in the enhanced quicksort. Each

routine was run for 100 times in each case and the average of the run times was obtained. The length of data was taken from 100 to 10000 when the first element or a random element was used as the pivot, and from 100 to 100000 when the pivot was the median of three. The timing was recorded using Java System.nanoTime() to record the time in nanoseconds, and the random number generator in Java rand.nextInt() was used for generating random numbers as data. As mentioned earlier the code for the classical and the enhanced quicksort algorithms was exactly the same except mainly the line where the pivot is to be determined. Moreover, all tests were performed on the same computer system.

We understand that as is the case in all empirical tests many factors play a role in determining the running time of an algorithm. For example, the hardware system being used, the operating system, the compiler and programming language used and its efficiency in handling recursive and non recursive calls, are factors among other factors that affect the running time of an algorithm. However, it should be evident from the above that we tried to minimize the effect of factors that affect the comparison between the algorithms. The random number generation allows us to compare algorithms on random data; however, most real data are not entirely random. Indeed, data in databases can actually be mostly sorted. To deal with this case we also investigated the performance of the algorithms when the data are sorted, reverse sorted.

The time the enhanced quicksort algorithm would take on already sorted data is of particular importance. This is due to the fact that its worst case running time is supposed to be $\theta(n \log n)$, but with a slightly greater constant factor.

In comparing the times taken by the algorithms we were interested in the ratio of the running time of the enhanced quicksort to that of the classical quicksort. Such a ratio can indicate whether the enhanced quicksort is an improvement over the classical quicksort with random and ordered data or not. For example, a ratio of 1 means both the enhanced and classical algorithms took the same amount of time to perform the sort. A ratio value less than 1 implies that the enhanced algorithm sorted the same data in less time than the classical quicksort. Similarly, a ratio value of more than 1 indicates that the classical quicksort outperformed the enhanced quicksort.

The tests were performed with various values of d and *lvl*. The range of values of d was from 2 to 11, and for *lvl* the range was from 0 to 4. It is worth reiterating that the value of *lvl* indicates the levels or the number of recursive calls we wait for before reverting to the ThinList function. For example,

when *lvl* is 0 we call the median of medians on the larger list whenever one of the lists is at least 10 times more than the other. On the other hand if the value of *lvl* is 1 then we don't call the ThinList function unless this is the second consecutive recursive call with the larger of the lists at least 10 times of the other. The value of *d* on the other hand gives the thinning factor for the list, since we only look at $\frac{n}{d^3}$ elements of the list for the median of medians.

3.2. *Results*

3.2.1. Performance Ratio Results

3.2.1.1. The pivot is the first item in the list

Pivot: First Item						**Data:** In increasing order				
	d=2	d=3	d=4	d=5	d=6	d=7	d=8	d=9	d=10	d=11
$t = 0$	0.1	0.07	0.06	0.05	0.06	0.06	0.07	0.08	0.09	0.09
$t = 1$	0.04	0.03	0.03	0.03	0.04	0.04	0.05	0.06	0.08	0.08
$t = 2$	0.04	0.03	0.03	0.03	0.04	0.04	0.05	0.06	0.07	0.07
$t = 3$	0.04	0.03	0.03	0.04	0.04	0.05	0.06	0.07	0.07	0.09
$t = 4$	0.04	0.03	0.03	0.05	0.04	0.06	0.06	0.07	0.08	0.09

Table 1: Average ratio for data whose size is in the range of 1000 to 7000. t is the threshold value

Pivot: First Item						Data: In random order				
	d=2	d=3	d=4	d=5	d=6	d=7	d=8	d=9	d=10	d=11
$t = 0$	2.371	1.633	1.457	1.345	1.284	1.266	1.241	1.248	1.220	1.190
$t = 1$	1.305	1.091	1.031	1.065	1.041	1.027	1.017	1.016	1.020	0.983
$t = 2$	1.118	1.040	1.003	0.965	1.022	1.007	0.959	1.001	0.990	0.999
$t = 3$	1.071	0.974	0.999	0.989	0.981	0.968	0.970	0.962	0.961	0.977
$t = 4$	0.976	0.969	0.999	0.958	0.965	0.965	0.968	0.965	0.965	0.970

Table 2: Average ratio for data whose size is in the range of 1000 to 7000. t is the threshold value

Tables 1 and 2 give us the average ratio of the performance of the enhanced quicksort algorithm to that of the classical quicksort algorithm for data size ranging from 1000 to 7000, when the pivot used is the first item in the list. Table 1 gives the performance ratio when the data is already sorted, and table 2 gives the performance when the data in the list is random. As expected the performance of the of the enhanced quicksort is superior to the classical quicksort

when the data is sorted, since as was explained in [6] the asymptotic bound is $O(n \log n)$ and it is $O(n^2)$ for the classical quicksort. Indeed as expected when the size of data increases the ratio becomes smaller and smaller and the ratio reaches 0.012 in the actual run time when the data is 7000 items in size.

When the data in the list is random, the picture is not as bright as in the first case. However, for threshold level of 2 or more, we see that the performance is about the same as that of classical quicksort, but becomes better for higher threshold levels. Indeed, for threshold level of 4, the enhanced quicksort is always slightly better than the classical quicksort.

Strangely, the increase in the threshold level has an effect on the thinning out factor *d*. As the threshold level increases the increase in d produces a parabolic shape that attains its minimum somewhere between the lowest and highest values of *d*. For example, when *d*=5 the enhanced quicksort is about 4% faster than the classical quicksort.

3.2.1.2. The pivot is a random item in the list

Pivot: Random element						Data: In increasing order				
	d=2	d=3	d=4	d=5	d=6	d=7	d=8	d=9	d=10	d=11
$t = 0$	1.90	1.47	1.42	1.48	1.54	1.59	1.68	1.73	1.84	1.90
$t = 1$	1.18	1.03	1.03	1.08	1.11	1.14	1.20	1.24	1.27	1.34
$t = 2$	1.03	0.95	0.95	0.93	0.94	0.94	0.94	0.94	0.95	0.95
$t = 3$	0.97	0.94	0.94	0.94	0.93	0.93	0.93	0.94	0.93	0.93
$t = 4$	0.96	0.94	0.92	0.93	0.93	0.93	0.93	0.93	0.93	0.93

Table 3: Average ratio for data whose size is in the range of 1000 to 6000.
t is the threshold value

Pivot: Random element						Data: In random order				
	d=2	d=3	d=4	d=5	d=6	d=7	d=8	d=9	d=10	d=11
$t = 0$	1.29	1.06	1.00	0.99	0.97	0.96	0.96	0.96	0.95	0.96
$t = 1$	0.98	0.92	0.91	0.90	0.89	0.90	0.89	0.91	0.90	0.89
$t = 2$	0.92	0.90	0.91	0.90	0.90	0.89	0.89	0.89	0.90	0.89
$t = 3$	0.91	0.90	0.90	0.90	0.90	0.89	0.90	0.90	0.89	0.89
$t = 4$	0.90	0.90	0.90	0.90	0.89	0.89	0.90	0.89	0.89	0.89

Table 4: Average ratio for data whose size is in the range of 1000 to 6000.

Tables 3 and 4 above give the average ratio of the performance of the enhanced quicksort to the classical quicksort when the pivot is chosen as a random element in the list. The time improvement is not as bright as in the case when we choose the first element in a sorted list as in table 1, due to the fact that for sorted items, random pivots produce better asymptotic performance on the average for the classical quicksort than the a first element pivots. However, the enhanced quicksort quickly outperforms the classical one for $t = 3$ or above.

3.2.1.3. The pivot is the median of three.

Pivot: Median of three							Data: In increasing order			
	d=2	d=3	d=4	d=5	d=6	d=7	d=8	d=9	d=10	d=11
$t = 0$	6.34	3.15	2.32	2.26	2.40	2.73	3.12	3.61	4.07	4.64
$t = 1$	1.45	1.14	1.06	1.06	1.07	1.10	1.12	1.18	1.24	1.28
$t = 2$	0.98	0.95	0.94	0.96	0.94	0.95	0.95	0.94	0.96	0.97
$t = 3$	0.93	0.93	0.91	0.93	0.94	0.93	0.94	0.92	0.91	0.93
$t = 4$	0.93	0.92	0.91	0.94	0.92	0.93	0.92	0.92	0.92	0.92

Table 5: Average ratio for data whose size is in the range of 1000 to 100000.

Pivot: Median of three							Data: In random order			
	d=2	d=3	d=4	d=5	d=6	d=7	d=8	d=9	d=10	d=11
$t = 0$	2.34	1.52	1.24	1.16	1.13	1.08	1.07	1.06	1.04	1.04
$t = 1$	0.93	0.88	0.90	0.89	0.90	0.90	0.91	0.88	0.90	0.88
$t = 2$	1.24	0.91	0.87	0.87	0.90	0.88	0.89	0.88	0.89	0.88
$t = 3$	1.20	0.90	0.87	0.89	0.88	0.91	0.91	0.90	0.89	0.88
$t = 4$	1.21	0.90	0.88	0.87	0.90	0.90	0.90	0.87	0.90	0.88

Table 6: Average ratio for data whose size is in the range of 1000 to 100000.

Again as expected the performance of classical quicksort improves when the pivot items is not always the first element in the list but the median of three items in the list. In such a case, and in particular for the threshold level t=0, on one hand, the median algorithm is called to find the pivot of the larger list almost always, since we have an already sorted list as input. On the other hand, in most cases the classical quicksort would pick an item that is very close to the median if not the median already, since the list is sorted or almost sorted. Thus, we see that the enhanced

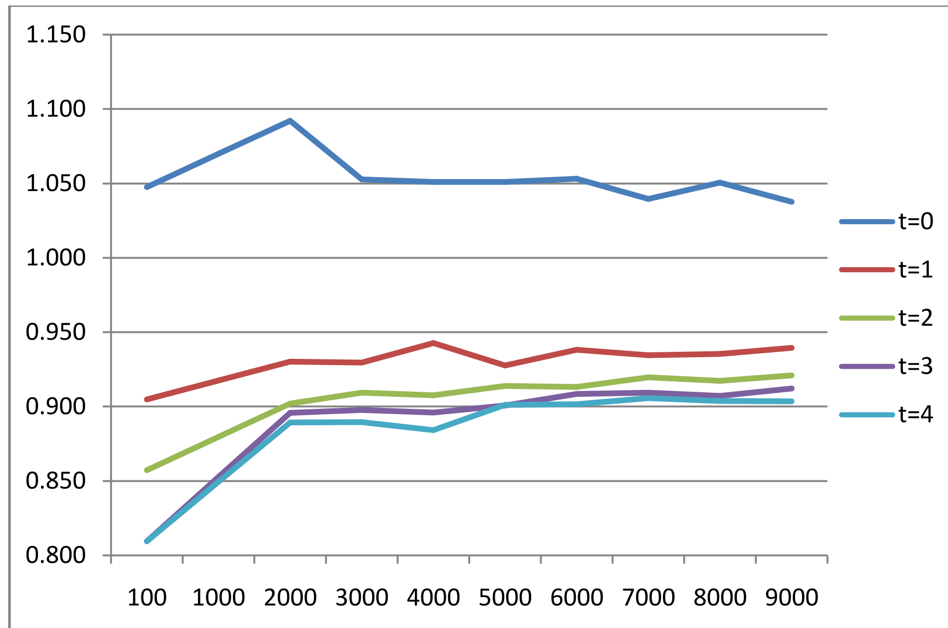
quicksort performs poorly compared to the classical quicksort in this case. However, when the threshold level is increased to 2 or more and for d=3 or more we see that the enhanced quicksort comfortably outperforms the classical one, whether the data is already sorted or completely random. Indeed, for properly chosen values of lvl and d, we get an improvement of 12% or more in performance.

As one final point, in this case and unlike the previous cases, we managed to go up to sizes of 100000 without exceeding the time limits in java, which is an indication of how superior is the median of three pivots to single element pivots.

3.2.1.3.1. Choosing d=ln(n).

Pivot: Median of three				Data: In random order				***d***= *ln*(*n*)		
n	100	1000	2000	3000	4000	5000	6000	7000	8000	9000
t=0	1.048	1.070	1.092	1.053	1.051	1.051	1.053	1.040	1.051	1.038
t=1	0.905	0.917	0.930	0.929	0.943	0.928	0.938	0.934	0.935	0.940
t=2	0.857	0.880	0.902	0.909	0.908	0.914	0.913	0.920	0.917	0.921
t=3	0.810	0.853	0.896	0.898	0.896	0.901	0.908	0.909	0.907	0.912
t=4	0.810	0.849	0.889	0.890	0.884	0.901	0.901	0.906	0.904	0.904

Table 2: Average ratio for data whose size (n) is in the range of 100 to 9000, and the list thinning factor d= ln(n). t is the threshold value



When the thinning out factor d is chosen to be the natural logarithm ($\ln n$) of the size of the list, we can see for all threshold levels above t=0, the enhanced algorithms outperforms the classical quicksort. We get further improvements as the threshold level increases, but not at the same rate as going from t=0 to t=1 threshold levels.

4. Conclusions.

In this paper we studied the practicality of the enhanced quicksort algorithm of [6]. We presented results that show the effect of using the median of medians as a pivot in restricted manner can improve the running time of the quicksort algorithm available. The restriction was to revert to the median of medians algorithm only when there is a consistent recent history or "evidence" that the partitioning method would lead to uneven size lists. In our implementation we used 10 as a factor indicating that the lists are uneven, or in other words that one of the partitioned lists is least 10 times the size of the other. However, this factor needs to be studied further to get to a better definition of unevenness in partitioned lists size. We studied the effect of applying the median of medians on thinned out lists by a factor of 2^3, 3^3, 4^3, … , 11^3. However, our results show that we get better improvements by restricting the recursive calls to the median of medians than by thinning the list, but together they can give more practical solution to the classical quicksort.

Our study of the enhanced quicksort can be adopted to other variants like the dual pivots quicksort of Yaroslavskiy, since it is only applied to the larger of the partitioned lists when there

is a degree of persistence in getting uneven size lists. Indeed, we believe that the "controlled" use of the median of medians in quicksort can produce quite practical sorting algorithm even by today's standards.

References

[1] Williams, J. W. J. , *Algorithm 232 - Heapsort,* Comm. ACM, *7 (6): 347–348, 1964.*

[2] Knuth, Donald, *"Section 5.2.4: Sorting by Merging". Sorting and Searching.* The Art of Computer Programming. *3, 2nd ed., Addison-Wesley. pp. 158–168, 1998.* ISBN 0-201-89685-0.

[3] Hoare, C. A. R. *(1961). "Algorithm 64: Quicksort".* Comm. ACM. *4 (7): 321.*

[4] *Shell, D. L. (1959). "A High-Speed Sorting Procedure" (PDF). Communications of the ACM. 2 (7): 30–32.*

[5] Mirza Abdulla, *An efficient enhancement to selection sort, submitted.*

[6] Mirza Abdulla, *Quick Sort with Optimal Worst Case Running Time*, American Journal of Engineering Research (AJER), Volume-6, Issue-1, pp-32-36, 2017.

[7] Blum, M.; Floyd, R. W.; Pratt, V. R.; Rivest, R. L.; Tarjan, R. E. *Time bounds for selection*. Journal of Computer and System Sciences. *7* (*4*): *448–461*, August 1973.

[8] Sebastian Wild. 2013. *Java 7's Dual Pivot Quicksort*. Master's thesis. University of Kaiserslautern.

[9] Mirza Abdulla, *Selection Sort with Improved Asymptotic Time Bounds*, The International Journal Of Engineering And Science (IJES), *Vol 5, Issue 5, pp 125-130, 2016.*

[10] Mirza Abdulla, *Marrying Inefficient Sorting Techniques Can Give Birth to a Substantially More Efficient Algorithm*, International Journal of Computer Science and Mobile Applications, *Vol.3 Issue. 12, pp. 15-21, 2015.*

[11] Mirza Abdulla, *An O(n^(4/3)) Worst Case Time Selection Sort Algorithm* , International Journal of Computer and Electronics Research, *Vol 5, Issue 3, pp. 36-41, 2016.*