Cumhuriyet Üniversitesi Fen Fakültesi
Fen Bilimleri Dergisi (CFD), Cilt:36, No: 6 Özel Sayı (2015)
ISSN: 1300-1949

Cumhuriyet University Faculty of Science
Science Journal (CSJ), Vol. 36, No: 6 Special Issue (2015)
ISSN: 1300-1949

Enhancing and Optimization Sorting Algorithms: An Empirical Study

Mohammad Mehdi KARIMIZADEH[1,2], Ehsan RAFEAZADEH[1,2], Pouran AMIRI[1,2],

Dariush KHOLGHNIK[3]

[1]*Department of Computer Engineering, Fars Science & Research Branch, Islamic Azad University, Marvdasht, Iran*

[2]*Department of Computer Engineering, Marvdasht Branch, Islamic Azad University, Marvdasht, Iran*

[3]*The Expert of Informatics, IRIB Yasouj Center, Yasouj, Iran*

Received: 20.04.2015; Accepted: 09.07.2015

Abstract. Sorting algorithms are used to sort a list of data. Also sorting is used in other computer operations such as searching, merging, and normalization. Since the sorting is considered as a one of the key operation in computer science, recognition of an optimization approaches can develop this science considerably. Optimization in the sorting algorithms, even in small scale, can cause saving a lot of time. The main discussion of the paper is on those algorithms which present optimized versions of classical sort algorithms. We studied classical and optimized methods some of sorting algorithm such as Selection sort, Bubble sort, Insertion sort, Quick sort and Heap sort and compared each algorithm in terms of the running times when used for sorting arrays of integers.

Keywords: Sorting Algorithm, Classic Sorting Method, Optimized Sorting Method, Optimization algorithm

1. INTRODUCTION

Sorting is considered one of the basic operations in computer sciences. The goal of sorting is create a list of data with specific sequence. Sorting is help to operation that need to be sorted data, such as searching, merging and normalization. There are several methods for sorting algorithm. After selecting one of the methods, programmer writes codes that have high performance, because sorting might be the main and important section of the program and any improvement in sorting speed leads to the run faster of program. Another reason is that most sorting algorithms are frequently used in the programs and optimization can cause effect on performance of system [7].

Researchers' interest to sort goes back to more than one century ago. In 1890, Hollerith develops systems which include punch, charting and sorting. The issue of sort algorithm goes back to 1940s. In 1945, merge sort presented by John von Neumann, the Hungarian-American mathematician, is considered to one of the first sorting algorithm in computer science. In 1950s and 1960s, other sorting algorithms are represented. For example, quick sort algorithm is proposed by Tony Hoare in 1961. In later years of 1960s, development of sorting algorithms was studied. For example, development of quick sort algorithm, Scowen and Singleton algorithms are proposed in 1969. In later years of 1980s, optimization of sorting algorithm is emphasized; the issue which is being focused even today.

In the paper, classic sorting algorithms and their optimization methods will be studied. In section 2, we studied five sorting algorithms, including selection sort algorithm, bubble sort algorithm,

* Corresponding author. *E-mail:* Karimizadeh_mehdi@yahoo.com

Special Issue: International Conference on Non-Linear System & Optimization in Computer & Electrical Engineering

insertion sort algorithm, quick sort algorithm and heap sort algorithm. During the studies, classical and optimized versions of algorithms are explained. In section 3, experimental results of classical and optimized algorithms are analyzed and finally, section 4 will be the conclusion this paper. It should be noted that all the represented examples of the present study are discussed on the basis of ascending sorting.

2. SORTING ALGORITHMS

In this section, we studied classical and optimized methods of common sorting algorithms, including selection sort, bubble sort, insertion sort, quick sort and heap sort.

2-1 Selection Sort

2-1-1 Classical Method

In this method, the element with maximum value is determined and exchanged with last element. In the next step, the process is repeated for other elements and the element with maximum value is exchange with one before the last element. Therefore, at the end of each step, one element is located at their right place. The algorithm continues until all elements in the list will be sorted [5]. The classical selection sort algorithm is listed below.

1. Repeat steps 2 to 5 until **length=1**
2. Set **max=0**
3. Repeat for **count=1** to **length**
   If (**a[count]>a[max]**)
      Set **max=count**
   End if
4. Interchange data at location **length-1** and **max**
5. Set **length=length-1**

Here ***a*** is the unsorted input list and ***length*** is the length of array. For example, According to Table 1, 91 (91 is the element with maximum value) is exchanged with 13 (13 is the last element). In the second step, 82 (82 is the element with maximum value excluding 91) is exchanged with one before the last element that here is the same 82. The algorithm continues until all elements in the list will be sorted.

Table 1. An Example of Classical Selection Sort Algorithm.

Primary List	45	3	91	67	82	13
Step 1	45	3	13	67	82	91
Step 2	45	3	13	67	82	91
Step 3	45	3	13	67	82	91
Step 4	13	3	45	67	82	91
Step 5	3	13	45	67	82	91

2-1-2 Optimized Method

This method is proposed on the basis of remove of some of unnecessary comparison instructions. In this method, after find of the new maximum element, the place of former maximum element is saved in the stack. Therefore, in the next step, does not require comparing elements before place of the former maximum element and start a new step will be the place of the former maximum element. The optimized selection sort algorithm is listed below [3].

1. Repeat steps 2 to 9 until **length=1**
2. If stack is empty
   Push 0 in the stack
   End if
3. Pop stack and put in **max**
4. Set **count=max+1**
5. Repeat steps 6 and 7 until **count<length**
6. If (**a[count]>a[max]**)
   a. Push **count-1** on stack
   b. Interchange data at location **count-1** and **max**
   c. Set **max=count**
   End if
7. Set **count=count+1**
8. Interchange data at location **length-1** and **max**
9. Set **length=length-1**

Here ***a*** is the unsorted input list and ***length*** is the length of array. For example, According to Table 2, 45 is compared with 3 and determined the 45 as a maximum element. The list will continue until we find 91 (91 is new maximum). Here 45 is the former maximum and exchanged with 3 (3 is element before the new maximum). The list will continue again and since not found a new maximum, 91 exchanged with 13 (13 is the last element). Therefore, at the end of step 1, 91 are located at their right place and step 2 starts will be the place of 45 (45 is the former maximum element). The algorithm continues until all elements in the list will be sorted.

Table 2. An Example of Optimized Selection Sort Algorithm.

Iteration	List	Maximum	Former Maximum
Primary List	45,3,91,67,82,13	-	-
Step 1	3,45,13,67,82,91	91	45
Step 2	3,13,45,67,82,91	82	67
Step 3	3,13,45,67,82,91	67	45
Step 4	3,13,45,67,82,91	45	-
Step 5	3,13,45,67,82,91	13	-

2-2 Bubble Sort

2-2-1 Classical Method

In this method, any step includes several comparison in which the compared an element with the next element. If be more than the next element, then exchanged with it. Therefore, at the end of each step, maximum element is located at their right place. The algorithm continues until all elements in the list will be sorted [2]. The classical bubble sort algorithm is listed below.

1. Repeat step 2 for **i = 0 to length-2**
2. Repeat step 3 for **j= 0 to length-i-2**
3. If (**a[j]>a[j+1]**)
   Interchange a[j] and a[j+1]
   End if

Here ***a*** is the unsorted input list and ***length*** is the length of array. For example, According to Table 3, 45 is compared with 3 and since it is more than of it, the exchanged with 3. Then 45 are compared

with 91 and since it is less than of it does not require to exchange with 91. This process will continue until the end of the list. Therefore, at the end of step 1, 91 are located at their right place. The algorithm continues until all elements in the list will be sorted.

Table 3. An Example of Classical Bubble Sort Algorithm.

Primary List	45	3	91	67	82	13
	3	45	91	67	82	13
	3	45	91	67	82	13
	3	45	67	91	82	13
	3	45	67	82	91	13
Step 1	3	45	67	82	13	91
	3	45	67	82	13	91
	3	45	67	82	13	91
	3	45	67	82	13	91
Step 2	3	45	67	13	82	91
	3	45	67	13	82	91
	3	45	67	13	82	91
Step 3	3	45	13	67	82	91
	3	45	13	67	82	91
Step 4	3	13	45	67	82	91
Step 5	3	13	45	67	82	91

2-2-2 Optimized Method

This method is almost the same as optimized selection sort method and the only difference is in the type of sorting method that here exchange elements are according to the bubble sort. The optimized bubble sort algorithm is listed below [3].

1. Set **top = -1**
2. Repeat steps 3 to 5 for **i = 0 to n-2**
3. If (**top<0**)
   Push 0 on stack
   End if
4. Pop stack and put in **val**
5. Repeat step 6 for **j = val to n-i-2**
6. If (**a[j]>a[j+1]**)
   Interchange **a[j]** and **a[j+1]**
   Else
   Push **j** on stack
   End if

Here ***a*** is the unsorted input list and ***length*** is the length of array. For example, According to Table 4, 45 is compared with 3 and since it is more than of it, the exchanged with 3 (determined the 45 as a maximum element). The list will continue until we find 91 (91 is new maximum). Here 45 is the former maximum. The list will continue again and 91 is compared with 67 and since it is more than of it, the exchanged with 67. This process will continue until the end of the list. Therefore, at the end of step 1, 91 are located at their right place and step 2 starts will be the place of 45 (45 is the former maximum element). The algorithm continues until all elements in the list will be sorted.

Table 4. An Example of Optimized Bubble Sort Algorithm.

Iteration	List	Maximum	Former Maximum
Primary List	45,3,91,67,82,13	-	-
Step 1	3,45,67,82,13,91	91	45
Step 2	3,45,67,13,82,91	82	67
Step 3	3,45,13,67,82,91	67	45
Step 4	3,13,45,67,82,91	45	-
Step 5	3,13,45,67,82,91	13	-

2-3 Insertion Sort

2-3-1 Classical Method

In this method, first and second elements are compared with each other and they are sorted. Then the third element is added to them, So that three elements are sorted. Similarly, the next elements are added to the list and are sorted [9]. The classical insertion sort algorithm is listed below.

1. Repeat steps 2 to 5 for **i=1 to n-1**
2. Set **temp = a[i]**
3. Set **j=i-1**
4. Repeat steps 4a and 4b while (**temp<a[j] and j>=0**)
   4a. Set **a[j+1] = a[j]**
   4b. Set **j = j-1**
5. Set **a[j+1] = temp**

Here ***a*** is the unsorted input list and ***length*** is the length of array. For example, According to Table 5, at the step 1, 11 and 6 is compared and inserted so that to be sorted. In step 2, 97 are added to them, So that three elements are sorted. The algorithm continues until all elements in the list will be sorted.

Table 5. An Example of Classical Insertion Sort Algorithm.

Primary List	11	6	97	13	27
Step 1	6	11	97	13	27
Step 2	6	11	97	13	27
Step 3	6	11	13	97	27
Step 4	6	11	13	27	97

2-3-2 Optimized Method

In classical method, element insertion and sort are the very costly. Optimized insertion sort algorithm inserts new elements from both sides of list. As a result, we can reduce the cost to insert a new element, by reducing the number of shifts. The only problem emerges in the fact that due to insertion and shifting of new elements from both sides of list, the list may not be empty for element insertion. This means that on one side the space is empty and on the other side of space is full and to insert a new element is not empty place. To solve this problem, creates a temporary list, while its size is double the size of the main list and elements are inserted in the temporary list, So that the first element is inserted in place of the middle temporary list and the next elements inserted on the left or right

side by shift. After the sorting, sorted elements are transformed from the temporary list to the main list. The optimized insertion sort algorithm is listed below [3].

1. Set **left = length**
2. Set **right = length**
3. Set **b[left] = a[0]**
4. Repeat steps 5 to 9 for **i = 1 to length-1**
5. If (**a[i]>=b[right]**)
   5a. Set **right = right+1**
   5b. Set **b[right]=a[i]**
   5c. Go to step 4
   End if
6. If(**a[i]<=b[left]**)
   6a. Set **left = left-1**
   6b. Set **b[left] = a[i];**
   6c. Go to step 4
   End if
7. **Set loc = right**
8. Repeat while (**a[i]<b[loc]**)
   Set **loc = loc-1**;
9. If (**right-loc<loc-left**)
   9a. Set **j=right+1**
   9b. Repeat steps 9bx and 9by while (**j>loc+1**)
      9bx. Set **b[j]=b[j-1]**
      9by. Set **j=j-1**
   9c. Set **right=right+1**
   9d. Set **b[loc+1]=a[i]**
   Else
   9a. Set **j=left-1**
   9b. Repeat step 9bx and 9by while (**j<loc**)
      9bx. Set **b[j]=b[j+1]**
      9by. Set **j=j+1**
   9c. Set **left = left-1**
   9d. Set **b[loc] = a[i]**
   End if
10. Repeat steps 10a and 10b for **i = 0 to n-1**
   10a. Set **a[i]=b[left]**
   10b. Set **left = left+1**

Here ***a*** is the unsorted input list and ***length*** is the length of array. For example, According to Figure 1, initially 11 inserted in place of the middle temporary list. 6 is the next element and since 6 is less than 11, temporary list be shifted one times to the right and inserted before the 11. 97 is the next element and since 97 is more than 11, temporary list be shifted two times to the left and inserted after the 11. The algorithm continues until all elements in the list will be sorted. Ultimately, sorted elements are transformed from the temporary list to the main list.

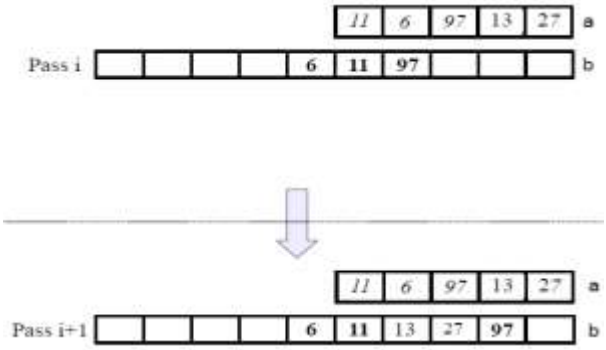


Figure 1. An Example of Optimized Insertion Sort Algorithm.

2-4 Quick sort

2-4-1 Classical Method

This algorithm is based on Divide and Conquer method and known as Hoare's method. In this method, the first element of the list call pivot and divides a list into two sub-lists, So that the left sub-list elements are less than pivot and the right sub-list elements are more than pivot. Then is sorted each sub-list by recursive method [1]. The classical quick sort algorithm is listed below.

```
Procedure Quick sort (Var x: Array List; left, right: integer;(
Var i,j,pivot: integer;
Begin
        If left<right then begin
           i:=left;   j:=right+1;  pivot:=x[left];
           Repeat
                   Repeat
                           i:=i+1;
                   Until x[i]>=pivot;
                   Repeat
                           j:=j-1;
                   Until x[j]<=pivot;
                   If i<j then swap (x[i],x[j]);
          Until i>=j;
          Swap (x[left],x[j]);
          Quick sort (x,left,j-1);
          Quick sort (x,j+1,right);
        End; {if}
End;
```

Here ***x*** is the unsorted input list. For example, According to Table 6, 57 (as the first element of the list) is considered as pivot. The pointer moves from left to right and determines 77 (77 is more than the pivot). Another pointer moves from right to left and determines 31 (31 is less than the pivot). 77 and 31 is exchanged. The process continues and determines 86 and 31. But two pointers have crossed from each other, so 31 is exchanged with 57 (57 is pivot element). At the end of the step 1, pivot is located at its right place, so that pivot is more than left sub-list elements and less than right sub-list elements. The algorithm continues for each sub-list until all elements in the list will be sorted.

Table 6. An Example of Classical Quicksort Algorithm.

Primary List	57	12	30	40	77	86	31
	57	12	30	40	31	86	77
Step 1	31	12	30	40	**57**	86	77
Step 2	30	12	**31**	40	**57**	86	77
Step 3	12	**30**	**31**	40	**57**	86	77
Step 4	**12**	**30**	**31**	40	**57**	86	77
Step 5	**12**	**30**	**31**	**40**	**57**	86	77
Step 6	**12**	**30**	**31**	**40**	**57**	77	**86**
Step 7	**12**	**30**	**31**	**40**	**57**	**77**	**86**

2-4-2 Optimized Method

This method (known as Qsorte) is similar to the classic method. The difference is that the median element is considered as the pivot. Also, at each step, checked that the sub-lists are sorted or not? If a sub-list was sorted, in later steps are not needed to sorting again. But if not sorted, the process continues until a sub-list is sorted [4]. The optimized quick sort algorithm is listed below.

```
void Qsorte (int m, int n)
{
      int k, v;
     bool lsorted, rsorted;
      if (m < n)
         {
           FindPivot (m, n, v);
           Partition (m, n, k, lsorted, rsorted);
           if (! lsorted) Qsorte(m, k-1);
           if (! Rsorted) Qsorte(k, n);
          }
}
```

For example, According to Table 7, 40 (as the median element) are considered as the pivot. The pointer moves from left to right and determines 57 (57 is more than the pivot). Another pointer moves from right to left and determines 31 (31 is less than the pivot). 57 and 31 is exchanged. Then checked that sub-list elements before the place 31 are sorted or not? The answer is Yes, so in the next step, does not require to sorting it. Also checked that sub-list elements after the place 57 are sorted or not? The answer is No, so in the next step, the process continues until a sub-list is sorted. The algorithm continues until all elements in the list will be sorted [10].

Table 7. An Example of Optimized Quicksort Algorithm.

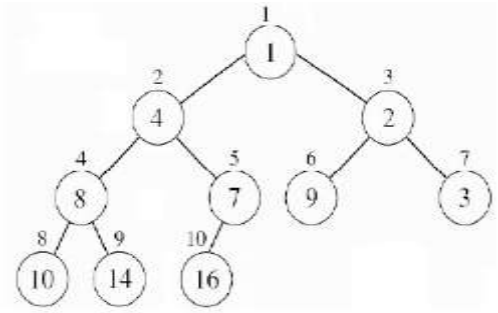
Primary List	12	30	57	40	77	86	31
Step 1	12	30	31	40	77	86	57
Step 2	12	30	31	40	77	57	86
Step 3	12	30	31	40	57	77	86

2-5 Heap sort

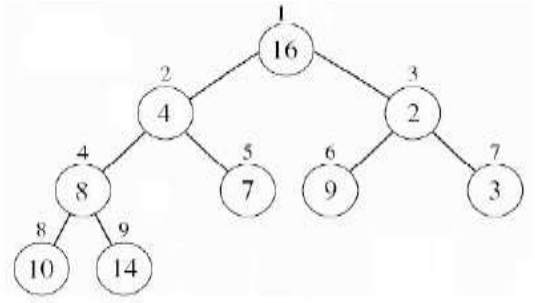
2-5-1 Classical Method

In this method, elements are implemented in a heap tree. Heap tree have the two modes; maxheap and minheap. In the present study, minheap and ascending array are considered as the study standards. In minheap, all Childs are bigger than their roots. This included is tree and sub-trees. To sorting, initially is removed root of the tree and is considered as the first element of the list. Then inserted is last node in the tree instead of the root and restructuring is tree. Again, the root is removed and inserted in the list and the restructuring operations are performed. The algorithm repeats until all elements in the list will be sorted [8].

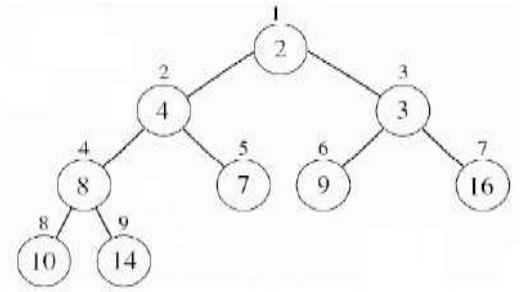
For example, According to Figure 2, 1 (1 is root element) is removed and inserted at the beginning of the list. Then, 16 are replacing instead of the root and restructuring is tree. In order to restructure, 16 are compared with 2 and 4 (2 and 4 are Childs 16) and exchanging with 2, because 2 is less than 4. Ultimately, 16 are exchanged with 3 and tree turns to heap. The algorithm repeats until all elements in the list will be sorted.



(a)



(b)



(c)

Figure 2. An Example of Classical Heap sort Algorithm: **(a)** The initial heap tree **(b)** Heap tree after removal element 1 and replace with 16 **(c)** Heap tree after restructuring.

2-5-2 Optimized Method

In this method, after removing the root, we have two heap sub-trees that is selected next element to enter on the list by comparing between its roots. For example, According to Figure 3, after removing the element 1 from tree and enter on the list, we choose element 2 among the sub-tree roots and enter on the list, because 2 is less than 4. Then 16 are replace instead 2 and restructuring is tree. The process is repeated until all elements in the list will be sorted [6].

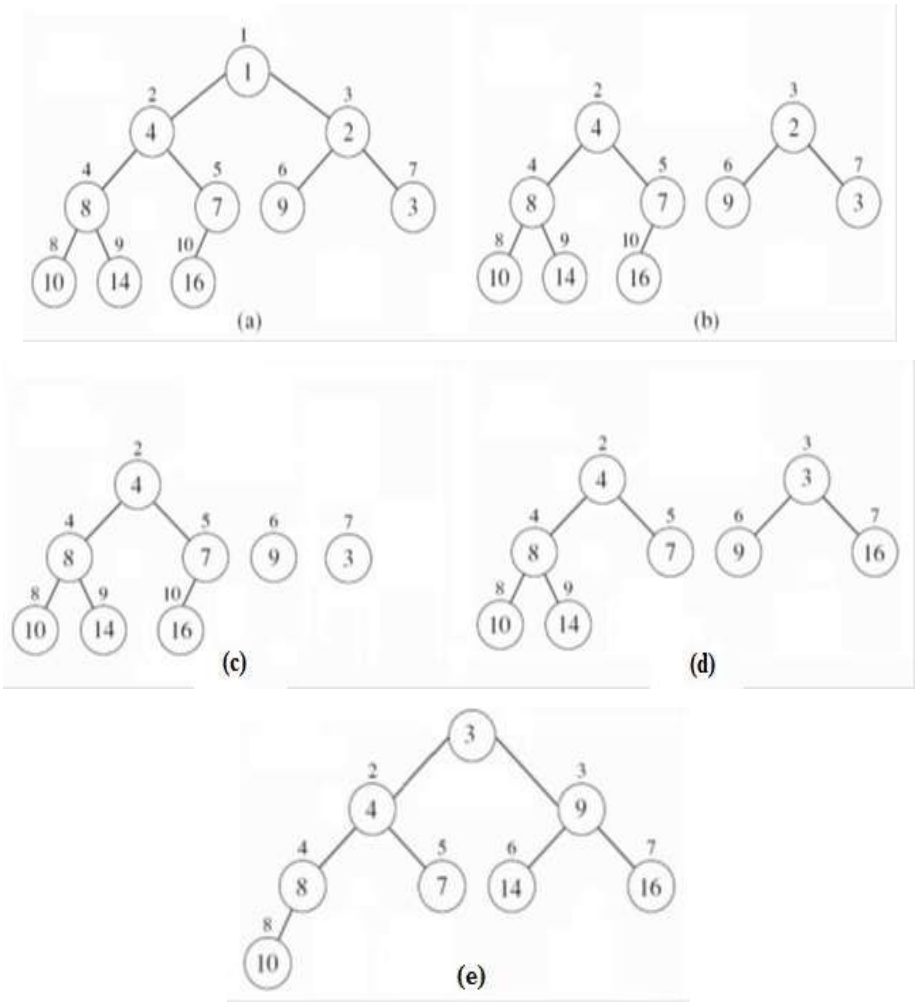


Figure 3. An Example of Optimized Heap sort Algorithm: **(a)** The initial heap tree **(b)** Heap tree after removal element 1 **(c)** Heap tree after removal elements 1, 2 **(d)** Heap tree after replace with 16 **(e)** Heap tree after restructuring.

3. EXPERIMENTAL RESULTS

In this section, we analyze the performance optimized methods on data lists and compare it with the classic methods. Analysis was performed in the same condition and on a 2.4 GHz Intel Core 2 Duo

processor with 2 GB 1067 MHz DDR3 memory machine with OS X version 10.6.8 platform. The figures 4, 5, 6 and 7 show the results of this analysis for classical and optimized methods of selection sort algorithm, bubble sort algorithm, insertion sort algorithm and quicksort algorithm, respectively.

In the selection sort and bubble sort, according to Figures 4 and 5, both classical and optimized algorithms results are almost the same for the low number of data. However, with the increasing number of data, Optimized methods improve more than double compared to classical method. In the quicksort, according to figure 7, with the increasing number of data, the effect optimized methods is higher compared with the classic methods. In the broad sense, in all the proposed sorting algorithms, Run-time Optimized methods are less Run-time classical methods.

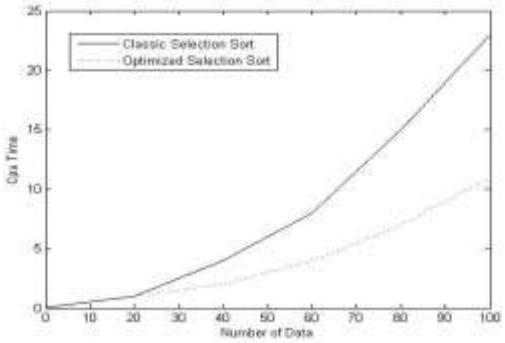


Figure 4. Compare the performance of Classical and Optimized Selection Sort Algorithm.

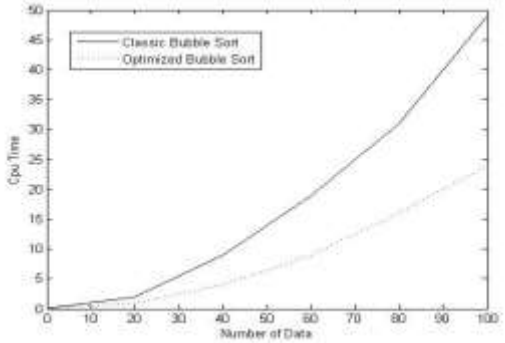


Figure 5. Compare the performance of Classical and Optimized Bubble Sort Algorithm.

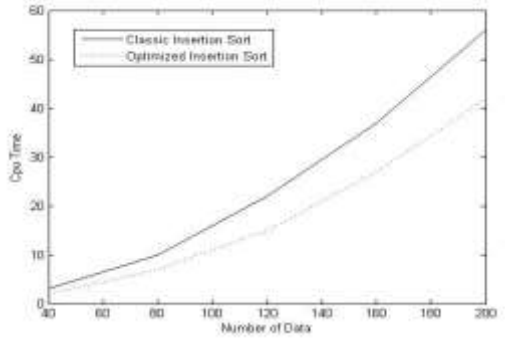


Figure 6. Compare the performance of Classical and Optimized Insertion Sort Algorithm.

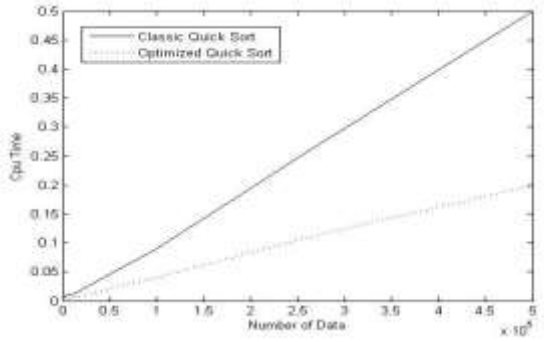


Figure 7. Compare the performance of Classical and Optimized Quicksort Algorithm.

4. CONCLUSION

This paper shows advantages of the proposed methods with the five major simple and comparative algorithms selection sort, bubble sort, insertion sort, quick sort and heap sort. The proposed optimized method has a significant improvement over the classical algorithms. To achieve this improvement, each method uses special features which shown in Table 8.

Table 8. A Compare of Optimized Sorting Algorithm.

Name	Average Case	Worst Case	Method	Special Feature
Selection Sort	$O(n^2)$	$O(n^2)$	Selection	Removed Some Useless Comparisons
Bubble Sort	$O(n^2)$	$O(n^2)$	Exchange	Removed Some Useless Comparisons
Insertion Sort	$O(n^2)$	$O(n^2)$	Insertion	Reducing Shift Operations
Quick Sort	O(n log n)	$O(n^2)$	Partition	Removed Some Useless Comparisons
Heap Sort	O(n log n)	O(n log n)	Selection	Removed Some Useless Comparisons

Other popular sorting algorithms from these categories have the potentiality to be improved by using the proposed approaches. Such as the merge sort, the radix sort, the shell sort and etc. These methods and the mentioned algorithms deserve future research.

5. REFERENCES

[1] Al-Kharabsheh, K. S., AlTurani, I. M., AlTurani, A. M. I., & Zanoon, N. I. (2013). Review on Sorting Algorithms A Comparative Study. International Journal of Computer Science and Security (IJCSS), 7(3), 120-126.

[2] Alnihoud, J., & Mansi, R. (2010). An Enhancement of Major Sorting Algorithms. The International Arab Journal of Information Technology, 7(1), 55-62.

[3] Khairullah, M. (2013). Enhancing Worst Sorting Algorithms. International Journal of Advanced Science and Technology, 56, 13-26.

[4] Khreisat, L. (2007). Quicksort A Historical Perspective and Empirical Study. International Journal of Computer Science and Network Security (IJCSNS), 7(12), 54-65.

[5] Kumar Karunanithi, A. (2014). A Survey, Discussion and Comparison of Sorting Algorithms. Master's Thesis, Department of Computing Science Ume'a University, Sweden.

[6] Levendeas, D., & Zaroliagis, C. (2008). Heap sort using Multiple Heaps. 2nd Panhellenic Student Conference on Informatics, EUREKA 2008, Research Gate, Samos, Greece, 93-104.

[7] Mishra, A. D., & Garg, D. (2008). Selection of Best Sorting Algorithm. International Journal of Intelligent Information Processing, 2(2), 363-368.

[8] Rao, D. T. V. D., & Ramesh, B. (2012). Experimental Based Selection of Best Sorting Algorithm. International Journal of Modern Engineering Research (IJMER), 2(4), 2908-2912.

[9] Savina, & Kaur, S. (2013). Study of Sorting Algorithm to Optimize Search Results. International Journal of Emerging Trends & Technology in Computer Science (IJETTCS), 2(1), 204-207.

[10] Wainright, R. L. (1987). Quicksort algorithms with an early exit for sorted subfiles. Comm. ACM, 183 190.