Modified Selection Sort Algorithm Employing Boolean and Distinct Function in a Bidirectional Enhanced Selection Technique

Ramcis N. Vilchez

***Abstract*—A sorting algorithm is a step by step procedure in arranging items on the list in particular order (ascending or descending). Sorting is one of the important data structure concepts that play a vital role in computer systems, file management, memory management, and many real-life applications. Among the sort algorithm, selection sort is the simplest and very straightforward. However, selection sort is considered the second worst algorithm concerning time complexity for large data. Due to the lousy performance of selection sort on large data, several enhancements were developed to improve runtime complexity. These enhancements have a significant improvement on the runtime complexity of the classical selection sort. However, the procedures presented in all these enhancements can still lead to some unnecessary comparisons, swapping and iterations that cause poor sorting performance. This study focuses on finding a remedy on the identified problems of the selection sort such as runtime complexity and unstable sorting by modifying the selection sort algorithm. The modified algorithm was tested using various data to validate the performance. The result was compared with the other available sorting algorithms to validate running time complexity. The results show that the Modified Selection Sort Algorithm Employing Boolean and Distinct Function in a Bidirectional Enhanced Selection Technique has a significant runtime complexity improvement compared with the other sorting algorithms. This study has a significant contribution to the field of data structures in computer science.**

***Index Terms*—Algorithm, bidirectional sorting, flag, selection sort, sorting.**

I. Introduction

A. Background of the Study

A sorting algorithm is a step by step procedure in arranging items on the list in particular order (ascending or descending). Sorting is one of the important data structure concepts that play a significant role in computer systems, file management, memory management and many real-life applications [1]. Some sorting algorithm is simple and spontaneous while others are complex but perform faster. Some work better on a smaller number of data, and some are good for specific range, some are suitable for floating point numbers, some algorithms are used for a large list of data, while some are used if the list has recurring values [2]. Sorting algorithms are problem specific, meaning they perform well on some specific problem and do not work well for all problems [3]. Selection sort, Bubble sort, and Insertion sort are very simple algorithms having the runtime complexity of $O(n^2)$ making them impractical to use for large data. Other complex algorithms like quicksort and merge sort are using the divide and conquer technique. These algorithms perform faster, having the runtime complexity of $O(n \log n)$ [3].

Among the sort algorithm, selection sort is the simplest and very straightforward. It resembles human instinct in arranging items in particular order. However, selection sort is considered the second worst algorithm in terms of time complexity [3].

The selection sort works by searching for the maximum value in the list and interchanging it with the last element. Then it finds for the second maximum value excluding the last element which was already found during the first pass and interchanging it with the second to the last element. In every procedure, the list is shrunk by one element at the end of the list. This processing is continued until the list becomes of size one when the list becomes trivially sorted [2].

In each procedure, to look for the maximum value, the selection sort starts from the beginning of the list. It starts by considering the first element to be the maximum and checks every element in the list whether the current maximum is the maximum. If it finds a greater value, then it considers that value to be the new maximum [2].

B. Problem Statement

The selection sort can be the most popular sort algorithm because of its simple and straightforward steps that resemble human instinct in arranging items. However, the procedure involved in the selection sort algorithm causes the following identified problems:

1) Unnecessary comparisons and swapping that leads to huge running time.
2) Inability to detect already sorted list during the early iterations that cause unnecessary iterations.
3) Inability to detect duplication of items that causes unstable sorting and unnecessary comparisons, swapping and iterations.

C. Objectives of the Study

This study aims to modify the selection sort algorithm to improve the time complexity. Specifically, it seeks to do the following:

1) Eliminate the unnecessary comparisons and swapping

Manuscript received December 19, 2018; revised July 5, 2019.

Ramcis N. Vilchez is with the Technological Institute of the Philippines (TIP), 938 Aurora Blvd, Cubao, Quezon City, Metro Manila, Philippines (e-mail: ramcis_vilchez@umindanao.edu.ph).

using bidirectional Enhanced Selection Sort Algorithm.
2) Detect already sorted list during the early iterations to terminate the unnecessary iterations using a Flag or Boolean.
3) Identify distinct items and grouping them adjacently to have a stable sorting using a distinct function.
4) Compare the results of the modified selection sort with other existing sorting algorithms using various data to determine the best algorithm in terms of execution time.

D. *Significance of the Study*

The result of this study if proven to be the best algorithm will be used in all sorting applications. Further, the concept presented in this study will be a very significant contribution in the field of data structures in computer science.

E. *Scope and Delimitations*

This study focuses on finding a remedy on the identified problems of the selection sort such as runtime complexity and unstable sorting by modifying the selection sort algorithm. The modified algorithm will then be tested using various data to validate the performance. The result will also be compared with the other available classical and modified sorting algorithms to validate running time complexity and ranking of the proposed modified selection sort.

II. Theoretical Framework

A. *Review of Related Literature*

Due to the lousy performance of selection sort on large data, several enhancements were developed to improve runtime complexity. These enhancements have a significant improvement on the runtime complexity of the classical selection sort. However, the procedures presented in all these enhancements can still lead to some unnecessary comparisons, swapping and iterations that cause poor sorting performance.

B. *Bidirectional Selection Sort*

In the study of [4], the concept of bidirectional selection sort was introduced. The main idea is that successive elements are selected on both sides of the array and placed in their proper position. In this technique, the sorting will be done in a single pass by two ways. That is to find the minimum element from the list and interchanged with the first element. At the same time, it will look for the maximum element and interchanged with the last element. Bidirectional Selection sort algorithm performs better than the classical since it reduces the number of swaps. However, there are still grey areas in this algorithm, since it cannot detect an already sorted list. It will continue to execute and finish the iterations even with the already sorted list. Unnecessary comparisons, swaps, and iterations are still possible in this improved algorithm.

C. *Enhanced Selection Sort Algorithm (ESSA)*

The concept introduced by [3] called Enhanced Selection Sort Algorithm (ESSA) eliminates some unnecessary comparisons by memorizing the location of the previous maximum when the new maximum is found. A stack is utilized to store the positions of the past or local maximums, which can be utilized in later iterations. It is assured that no value in the list is larger than the previous maximum value before the location of the previous maximum. However, this approach can be further improved. My proposed enhancement of the selection sort is based on this idea. However, I will be using bidirectional to look for the maximum and minimum in both sides of the array to eliminate unnecessary comparisons, swaps, and iterations. It is guaranteed that in the range between the former maximum and the just found current maximum there is no value greater than the former maximum. Therefore, there is no need to go through this range. In the next iteration, it is now safe to start looking for the next maximum from the location of the current maximum. This concept saves searching time, and it works better than the other enhancements in the selection sort algorithm. However, unnecessary comparisons, swaps, and iterations are still possible. Further, this algorithm cannot detect already sorted list.

D. *Bi-directional Mid Selection Sort*

The study of [5] called Bi-directional mid selection sort algorithm is based on bidirectional. However, in this enhancement of selection sort algorithm, it will sort the data by selecting (maximum and minimum) elements by starting to look from middle to both sides in the selected list by reducing the size of the list from n to 2 with the decrement of two in the size. It has the two loops outer and inner loop. Outer loop manages the size of the list to be processed for searching the maximum, and minimum number and the inner loop is for finding the smallest and various number from the list selected by the outer loop.

E. *Optimized Selection Sort Algorithm (OSSA)*

Another Selection sort enhancement from the study of [6] called Optimized Selection Sort Algorithm (OSSA) is also based on the bidirectional selection sort. However, instead of finishing the iterations from the beginning to the last element, the iteration will end if it reaches the middle of the array. This concept saves some iteration time as compared with the classical selection sort and bidirectional selection sort.

F. *Both Ended Sorting Algorithm*

Another enhancement is both ended sorting algorithm by [7], which claimed to be faster than the bubble and other algorithms. This algorithm will compare from both ends (from right end as well as from left end). This enhancement is based on the bubble sort algorithm that will compare one element from the front end with one element of the rear end. If the front element is greater than the rear end, then it will swap the front element with the rear element. In the second iteration, two consecutive elements from the front end and rear end of the array are compared. Replacing of elements is done if required according to the order. Here four variables are taken which stores the position of two rights elements and two left elements which are to be sorted. This process will continue until the list is sorted.

G. *Enhanced Bidirectional Selection Algorithm*

An enhancement of Selection sort algorithm by [1] is called Enhanced Bidirectional Selection Sort. This algorithm will select two values, smallest from the front and largest from the rear and placing them in their respective locations.

The smallest will be placed in the first location while the largest in the last location of another array thus, reducing the number of passes by half the total number of elements as compared with classical selection sort. The said maximum and minimum will then be deleted from the original list thus reducing the comparison by the factor of two.

H. Double Ended Selection Sort Algorithm (DESSA)

Another modified Selection sort algorithm introduced by [2] is called Double Ended Selection Sort Algorithm. This enhancement works by sorting elements in the same array and finding the maximum element and minimum element, exchanging the largest element with the last element and minimum element with the first element and then decreases the size of the array by two for the next iteration.

I. Improving the Performance of Selection Sort Using a Modified Double-Ended Selection Sorting

The idea of [8] is also promising since it uses two elements for both smallest and largest elements in the list and compare each other and placing them in their respective places in the front and rear locations. The researcher claimed that about 25% to 35% improvement in terms of time complexity was noted.

J. Upgraded Selection Sort

The study of [9] upgraded the selection sort by searching the smallest and largest items simultaneously and placing them on their right locations. This study was able to improve the number of iterations of the classical selection from n-1 to n/2. However, the time complexity remains the same.

K. Improved Selection Sort Algorithm

The concept presented in a study of [10] utilizes a queue to store the locations of all values that are the same as the maximum value. This idea is only useful if the given unsorted list is with duplications. However, if the list is already distinct, then the modified selection sort presented in this study is as bad as the classical selection sort for large data.

L. Minimizing the Execution Time of Selection Sort Algorithm

The study of [11] on selection sort enhancement presented the concept of dividing the array into two by getting the mean after finding the smallest and largest elements. The elements that are smaller or equal to the mean are placed at the front. While all the elements larger than the mean are placed at the rear portion. This concept has a significant improvement in the time complexity. The author claimed that for an average case scenario, the time complexity of the modified selection sort is now $O(n)$ from $O(n2)$ of that of the classical selection sort algorithm.

M. New Approach for Dynamic Bubble Sort Improvement

The approach presented in the study of [12] utilizes a stack to store the previous largest element to eliminate the unnecessary comparisons in the classical bubble sort algorithm. The succeeding iterations begin the search of the largest from the location of the previous largest element and not from the beginning of the array. With this approach, significant improvement in terms of time complexity was noted. For an average case scenario, the time complexity is $O(n2/4)$ compared to the classical bubble sort algorithm that has an $O(n2)$.

N. Insertion Sort with its Enhancement

An insertion sort enhancement approach presented in the study of [13] uses a bidirectional technique. For the first iteration, the first and the last element of the array is compared. If the first element is bigger than the last element, then the two elements are swapped. The location of the element from the left end and the element from the right end of the array are stored in the variables which are increased (left end) and decreased (right end) as the algorithm progresses. In the second iteration, two adjacent elements from the left of the array are taken and are compared. Insertion of elements is done if required according to the order. Then the similar process is carried as in Insertion sort. This approach is more efficient than the classical insertion sort algorithm.

O. Enhanced Insertion Sort Algorithm

Another insertion sort technique presented in the study conducted by [14] uses a bidirectional approach in sorting the list. Both sides of the array will be sorted accordingly depending on the sort order. If the algorithm sorts ascendingly, the small elements are inserted into the front portion. While the large elements are inserted in the rear portion. This approach has improved the time complexity of the insertion sort from $O(n2)$ to $O(n1.5)$ for an average case scenario.

P. Concept of the Study

The proposed modified selection sort algorithm as shown in Fig. 1, will be utilizing a stack to store the previous maximums or minimums. The positions of the values are stored in the list instead of storing the actual values. A distinct function is used to sort distinct elements only. A flag will also be utilized to determine a swap. If no swap detected during a pass, then the iteration will stop, and the list is already sorted.

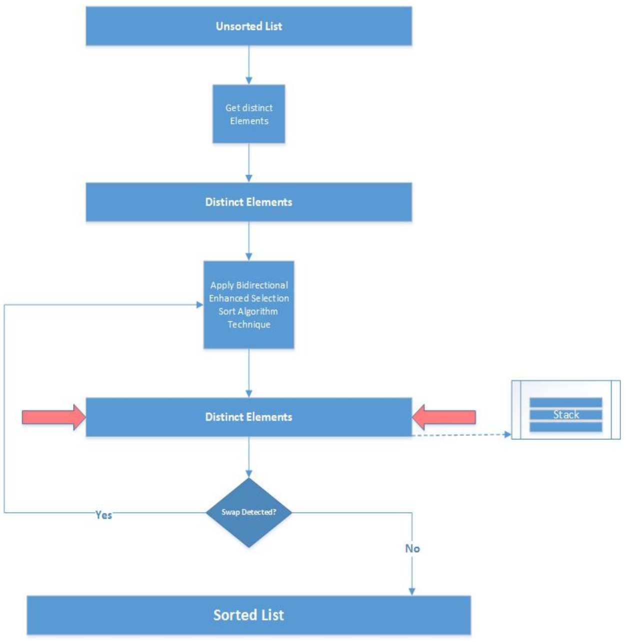


Fig. 1. Modified selection sort algorithm.

III. Operational Framework

A. Methods

This study aims to identify limitations or problems of selection sort algorithm and to find a remedy to these problems to improve the performance of selection sort. Let us first examine how selection sort works and determine its limitations or issues.

B. Classical Selection Sort Algorithm

The classical selection sort algorithm below works by searching for the maximum value in the list and interchanging it with the last element. Then it finds for the second maximum value excluding the last element which was already found during the first pass and interchanging it with the second to the last element. In every step, the list is reduced by one element at the end of the list. This procedure is continued until the list becomes of size one when the list becomes trivially sorted [2].

In each step, to find for the maximum value, the selection sort starts from the beginning of the list. It starts considering the first element to be the maximum and checks every element in the list to check if the current maximum is really the maximum. If it finds a greater value, it takes that value to be the new maximum [2].

This procedure is straightforward and easy to comprehend but has a lot of flaws that make it the second worst sorting algorithm for large items.

Algorithm: Selection Sort *(array[], length)* Here *L* is the unsorted input list and length is the length of an array. After the final execution of the algorithm array will become sorted. Variable max keeps the positions of the maximum value.

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

With the above algorithm, even with an already sorted list of items, the selection sort algorithm will still execute the nth iterations before arriving with a sorted list. The number of passes needed is still n-1 regardless of the list to be sorted. The procedure involved in the selection sort algorithm as illustrated above causes the following identified problems:

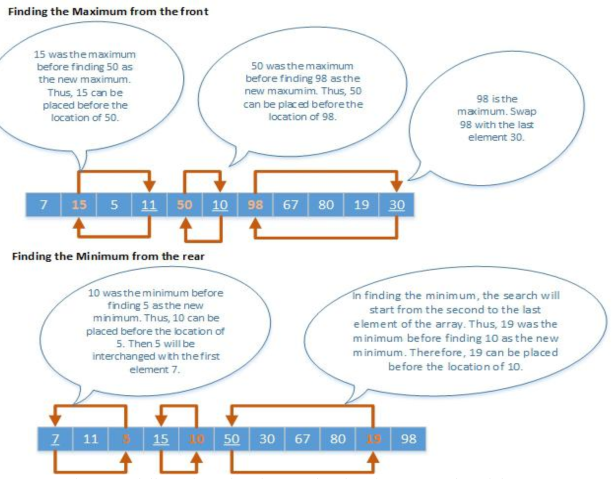
1) Unnecessary comparisons and swapping that leads to huge running time.
2) Inability to detect already sorted list during the early iterations that cause unnecessary iterations.
3) Inability to detect duplication of items that causes unstable sorting and unnecessary comparisons, swapping and iterations.

C. Proposed Enhancement of Selection Sort

To eliminate unnecessary comparisons, swaps, and iterations, a bidirectional Enhanced Selection Sort Algorithm will be used to memorize the location of the previous maximum (from left to right) and previous minimum (from right to left) when the new maximum and minimum are found. A stack is utilized to store the locations of the past or local maximums and minimums, which can be used in later iterations. It is assured that no value in the list is larger or smaller than the previous maximum and minimum value before the location of the previous maximum and previous minimum. Therefore, there is no need to go through this range. In the next iteration, it is now safe to start looking for the following maximum and minimum from the location of the current maximum and minimum. This concept saves searching time, and it works better than the other enhancements in the selection sort algorithm. This enhancement will solve problem number 1 in the enumerated identified problems above but cannot solve number 2 and 3.

To illustrate the proposed enhancement, for example, the list 7, 15, 5, 11, 50, 10, 98, 67, 80, 19, 30 is to be sorted. In this list, the maximum is 98, and it will be interchanged with the last value of the list, which is 30. On the other side, starting from the location of the second to the last item of the list, the minimum is 5, and it will be interchanged with the first value of the list which is 7. If the classical selection sort technique is followed, the list will be like 7, 15, 5, 11, 50, 10, 30, 67, 80, 19, 98 after the first pass. But the fact that before finding 98, the maximum value was 50 should be noticed, and likewise, before finding 5, the minimum was 10. So, it is assured that there is no value greater than 50 before the position of 50 and likewise, no value lesser than 10 before the position of 10 in the list. Therefore, we can start the next pass from the location of 50 for the maximum and 10 for the minimum, skipping other numbers in the list and removing some unnecessary searches. It is also observed that before finding 98 the maximum value was 50 and before finding 5 the minimum was 10. So, there is no value greater than 50 in the location range between 50 and the immediate past of the location of 98. Likewise, on finding the minimum, there is no value lesser than 10 in the location range between 10 and the immediate past of the location of 5. Subsequently, it is apparent that in the next iteration it is wastage of time to find for values greater than 50 and lesser than 10 before the current location of 98 and 5 respectively. Thus, the next iteration can start from the current position of the value 98 for the maximum and 5 for the minimum, reducing unnecessary comparisons. And 50, the former maximum, can be safely placed at the immediate past location of 98 by interchanging with the current value 10. Same with 10 the previous minimum can be safely placed at the immediate past location of 5. This approach leads to the list having the content 5, 11, 7, 10, 15, 19, 30, 67, 80, 50, 98 after the first iteration and now it possesses more degree of sorting, compared to the generated list using classical selection sort technique.

The second iteration looks for the second largest item and finds for a larger value than 50, starting from 30. It finds 67 to be the maximum and consider 50 to be the former maximum. Then 80 is found to be the new maximum and 67 to be the new former maximum. On the other side in finding for the minimum, the second iteration finds the second smallest item and looks for lesser value than 10, starting from the location of 7. It finds 7 to be the minimum and consider 10 to be the previous maximum. By following the same approach, after this iteration, the updated list is 5, 7, 11, 10, 15, 19, 30, 67, 50, 80, 98. In the third iteration, a larger value than 67 is looked for starting from the location of 50, which was the old location of the maximum 80 in the first iteration. Likewise, a smaller value than 10 is looked for starting at the position number 11. After the third iteration, the list will be 5, 7, 10, 11, 15, 19, 30, 50, 67, 80, 98. After the 3rd iteration, the list is already on its sorted order and after the 4th iteration, there is no swap detected. Thus, the iteration will stop. The proposed enhanced algorithm will be utilizing a flag to detect the occurrence of a swap. If there was no swap detected during each pass the iteration will stop with the assumption that the list is already sorted. To further elaborate on how the proposed enhancement work, consider the illustration in Fig. 2 below.


Finding the Maximum from the front
15 was the maximum before finding 50 as the new maximum. Thus, 15 can be placed before the location of 50.
50 was the maximum before finding 98 as the new maxumim. Thus, 50 can be placed before the location of 98.
98 is the maximum. Swap 98 with the last element 30.
7 15 5 11 50 10 98 67 80 19 30
Finding the Minimum from the rear
10 was the minimum before finding 5 as the new minimum. Thus, 10 can be placed before the location of 5. Then 5 will be interchanged with the first element 7.
In finding the minimum, the search will start from the second to the last element of the array. Thus, 19 was the minimum before finding 10 as the new minimum. Therefore, 19 can be placed before the location of 10.
7 11 5 15 10 50 30 67 80 19 98
Fig. 2. Bidirectional enhanced selection sort algorithm.

A Flag or Boolean will be utilized to determine a swap during an iteration. If no swap detected during an early pass, then the list is already sorted. This idea will solve problem number 2, that is the inability to detect already sorted list during the early iterations.

Finally, to solve the last problem number 3, that is detecting duplication, the researcher will be utilizing distinct function to make a stable sorting and to eliminate unnecessary comparisons, swaps, and iterations.

D. Modified Selection Sort Algorithm(MOSSA)

Here *L* is the unsorted input list, and length/n is the length of an array. After completion of the algorithm, the array will become sorted. Variable max keeps the position of the current maximum, while variable min keeps the location of the current minimum.

```
1. Get the distinct elements in the list.
2. Set Min=n-2
3. Repeat steps 4 to 25 while swap=true
4. Repeat steps 5 to 14 until length=1
5.  if stack is empty push 0 in the stack
6. Pop stack and put in max
7. Set count=max+1
8. Set swap=false
9.  Repeat steps 10 and 11 while count<length
10.if(L[count]>L[max])
        a. Push count-1 on stack
        b. Interchange data at location count-1
           and max
        c. Swap=true
        d. Set max=count
11.  Set count=count+1
12.  Interchange data at location length-1 and max
13. swap=true
14. Set length=length-1
15. Set i= 0 to n
16. Repeat steps 17 to 25 while i<n
17. if stack is empty push 0 in the stack
18. Pop stack and put in Min
19. Set countmin=Min-1
20. Repeat steps 21 and 22 until countmin< i
21. if(L[countmin]<L[Min])
        a. Push countmin+1 on stack
        b. Interchange data at location countmin+1 and
        Min
        c. swap=true
        d. Set min=countmin
22. Set countmin= countmin - 1
23. Interchange data at location i and min
24. swap=true
25. Set i=i+1
```

IV. Conclusion and Recommendations

Significant improvement with regards to the time complexity was noted based on the test results. The improvement can be attributed to the elimination of the identified problems from the procedure of classical selection sort and other sorting algorithms. The use of a flag to determine the already sorted list in early iteration and the utilization of distinct function play a vital role in the significant improvement of the modified algorithm.

Further improvement of this technique is recommended for future integration and application in the field of data warehousing and data mining.

Conflict of Interest

The author declares no conflict of interest.

Author Contributions

This research is solely conducted by the author below, including all analysis and other related task.

References

[1] J. Dua, "Enhanced bidirectional selection sort," *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, World Academy of Science, Engineering and Technology, vo. 8, no. 7, 2014.

[2] M. Khairullah, "An enhanced selection sort algorithm," *SUST Journal of Science and Technology*, vol. 21, no. 1, pp. 9-15, 2014

[3] M. Khairullah, "Enhancing worst sorting algorithms," *International Journal of Advanced Science and Technology*, vol. 56, July, 2014.

[4] R. M. Patelia, S. D. Vyas, P. S. Vyas, and N. S. Patel, "An enhanced selection sort algorithm," *International Journal of Advanced Technology in Engineering and Science*, vol. 3, no. 1, March 2015.

[5] M. F. Umar, E. U. Munir, S. A. Shad, and M. W. Nisar, "Enhancement of selection, bubble and insertion sorting algorithm," *Research Journal of Applied Sciences, Engineering and Technology*, vol. 8, no. 2, pp. 263-271, 2014.

[6] S. Jadoon, S. F. Solehria, S. Rehman, and H. Jan, "Design and analysis of optimized selection sort algorithm," *International Journal of Electric & Computer Sciences (IJECS-IJENS)*, vol. 11, no. 1, pp. 16-22, February 2011.

[7] A. Brijwal, A. Goel, A. Papola, and J. K. Gupta, "Both ended sorting algorithm & performance comparison with existing algorithm," *International Journal of IT, Engineering and Applied Sciences Research (IJIEASR)*, vol. 3, no. 6, June 2014.

[8] S. Lakra and Divya, "Improving the performance of selection sort using a modified double-ended selection sorting," *International Journal of Application or Innovation in Engineering & Management (IJAIEM)*, 2013.

[9] S. Chand, T. Chaudhary, and R. Parveen, "Upgraded selection sort," *International Journal on Computer Science and Engineering*, vol. 3, no. 4, 2011.

[10] J. B. Hayfron-Acquah, O. Appiah, and K. Riverson, "Improved selection sort algorithm," *International Journal of Computer Applications*, 2015.

[11] M. Kumar, M. Malhotra, and D. Ahuja, "Minimizing the execution time of selection sort algorithm," *International Journal of Engineering and Computer Science*, 2015.

[12] K. Thabit, E. Alsaggaf, and F. Bahareth, "New approach for dynamic bubble sort improvement," *Research Notes in Information Science*, 2013.

[13] P. K. Chhatwani "Insertion Sort with its Enhancement," *International Journal of Computer Science and Mobile Computing*, vol. 3, issue 3, March 2014.

[14] A. S. Mohammed, S. E. Amrahov, and F. V. Celebi, "Bidirectional Conditional Insertion Sort Algorithm; An efficient progress on the classical insertion sort," *Future Generation Computer Systems*, 2016.

Ramcis Vilchez got a bachelor of science in computer science from Mindanao State University, Marawi City. He finished his master's degree in information technology at Ateneo de Davao University, Davao City, Philippines. Currently, he is a graduation candidate for a degree in doctor of information technology in Technological Institute of Philippines-Quezon City, Philippines.

He is the current the dean of the College of Computing Education of the University of Mindanao, Davao City. He was a former president of the Council of Deans for Information Technology in Region XI from 2015 to 2017 and was reelected as a vice president for SY from 2017 to present. Further, he is an accreditor for Regional Quality Assurance of the Commission on Higher Education to evaluate Information Technology Education programs in Region 11.

He has several research presentations and publications in local and international research conferences.