ISSN 2347 - 3983
Volume 9. No. 6, June 2021
International Journal of Emerging Trends in Engineering Research
Available Online at http://www.warse.org/IJETER/static/pdf/file/ijeter04962021.pdf
https://doi.org/10.30534/ijeter/2021/04962021

# Analysis of MinFinder Algorithm on Large Data Amounts

**Wilson Philips[1], Wirawan Istiono[2]**
[1]Universitas Multimedia Nusantara, Indonesia, wilson4@student.umn.ac.id
[2]Universitas Multimedia Nusantara, Indonesia, wirawan.istiono@ umn.ac.id

## ABSTRACT

When dealing with large amounts of data, various sorting algorithms will be tested and searched for which algorithm is the most efficient. Many factors determine the level of performance of the sorting algorithm, such as time and size complexity, stability, accuracy, clarity, effectiveness, and so on. MinFinder is a newly discovered sorting algorithm by finding the smallest value in each iteration while the program is running. In this paper, the MinFinder algorithm will be tested on the structure of data arrays, vectors and linked lists to compare the speed of completion time. Based on the results of experiments on data with n amount of 10 power of 3, 10 power of 4, and 10 power of 5, it can be concluded that the best application of MinFinder is in the array, with the processing time needed 2X faster than other data structures. Vector and Linked Lists have weaknesses when accessing elements at each iteration, which makes them slower than arrays.

**Key words:** Linked List, MinFinder, Sorting, Time and Size Complexity, Vector.

## 1. INTRODUCTION

Sorting is a technique used to arrange data that is not sequential, from the smallest to the largest value, or vice versa [1]. When the data has been sequenced, the process of finding data will be easier to do. Many conventional sorting algorithms can be applied to sort data, such as Bubble Sort, Selection Sort, and Insertion Sort [2][3]. However, when implemented in large amounts of data, the algorithms take a long time to sort the data [4]. One of the newest sorting algorithms found is MinFinder.

The MinFinder algorithm is designed to find the smallest value in each iteration sorting an array or list and place it to the forefront by sliding the elements to the right. Additional memory is not required to perform the MinFinder algorithm and this algorithm is relatively stable because it does not change the position of the same element. The time complexity of the MinFinder algorithm is *O(n^2)*, and the size complexity is *O(1)* [5]**.**

## 2. OVERVIEW SORTING ALGORITHMS

There are two types of sorting algorithms in general, namely internal and external sorting. Internal sorting is done by storing all the elements that will be sorted in main memory, such as Bubble Sort, Selection Sort and Insertion Sort [6]. While external sorting is done by accommodating some portion of the elements in the secondary memory which is then transferred to the memory, then the results of the sorting will be stored in the secondary memory again [7]. Examples of external sorting algorithms are Merge Sort and Quick Sort. In designing a sorting algorithm, there are several properties that must be met [8], including:

- Input: The algorithm must have an input value of a defined set.
- Output: The algorithm produces output values that are defined as the solution of the problem to be achieved using the input provided.
- Definiteness: The steps of the sorting algorithm must be as detailed as possible to sort the elements.
- Correctness: The algorithm must produce output in the correct order, according to each given set of input elements.
- Finiteness: The algorithm must produce the desired output after a few calculated steps.
- Effectiveness: The algorithm should be designed considering the amount of time required.
- Generality: The algorithm must be able to be applied to every given problem, not only for certain sets

The MinFinder algorithm is included in the type of internal sorting algorithm, which does not require extra memory to do the sorting process [9]. Algorithm MinFinder can be done by finding the value of the smallest element of a list or array, then placed in the leading position, by shifting other elements to the right. In the sample case, the data need to sort descending, then the element that is placed at the front is the biggest element. Then in the second iteration, the smallest element will be searched for and then placed in the second leading position using the same method [4]. In Figure1 shown MinFinder pseudocode sorting algorithm.

```
L = A.length – 1,
NextIterPoint =0,
PositionOfMinValue = 0
Finder:
minValue = A[positionOfMinValue]
for i = positionOfMinValue + 1 to L
    if minValue > A[i]
        minValue = A[i]
        positionOfMinValue = i
        if i != L
            goto step2
    if I = L
        for j = positionOfMinValue to NextIterPoint
            A[j] = A[j-1]
        A[NextIterPoint] = minValue
        NextIterPoint++
        positionOfMinValue = NextIterPoint
        goto step2
    print(A)
```

**Figure 1:** MinFinder Pseudocode Algorithm

As shown in Figure 1, the step MinFinder Algorithm first step is initialize variables A [n], L = A.length () - 1, NextIterPoint = 0, PositionOfMinValue = 0; and after that enter a branching control to jump to a specific place and select the current element that holds the smallest value, then enter the MinValue variable; And after that iterates until the index on the array is smaller or equal to the length of the array, starting at the current MinValue position. After that, check each element in the array with MinValue, whether the MinValue value is greater or smaller than the current element. If the current MinValue element is greater than the element to the current index, then change the MinValue value to the value of that element and its position, to check the rest of the elements afterwards. MinValue = A [i]; PositionOfMinValue = i; Then check whether the current index is the last index of an array. If the current index is not the last index, then repeat to step 2. And then check whether the index of the element is now the last index of an array, to ensure the value of MinValue has been compared to all elements in the array. If true, then move the elements from the array one position to the right from the first element to the position of the smallest element, then the smallest element will be moved to the first position in the array. A[k] = A [k-1] where k = PositionOfMinValue to IterationPoint. And the last step, update the IterationPoint value and position of MinValue and then repeat to step 2, until all elements in the array have been sorted.

## 3. RESEARCH METHODOLOGY

In this study, the MinFinder algorithm will be tested on 3 different data structures, including arrays, vectors, and linked lists [10]. These three data structures are often found in computer science and each has advantages and disadvantages.

Array has the advantage of fast data access, because it can directly access the desired index. For data input, the array must be defined in advance how much data will be inputted, because the number of indexes in the array is static. Arrays do not have shift operations to shift a row of elements to the right or left, so that only the values of the elements can be overwritten.

Vector is a dynamic array, where the number of indexes will follow the amount of data inputted. Inputting data on the vector does not need to be defined in advance how much data [11]. Vector can be found in C ++, which has been optimized for application rather than arrays. STL operations on C ++ are used to access and change vector contents [12]. This operation makes it easy for data to be changed at certain locations, which is more practical to use than using arrays.

Linked List is a linear data structure, where each element will be allocated to heap memory. In a linked list, an element is like a struct, which has more than 1 member variable in it. Accessing this linked list cannot be done directly, it must go through head, tail, or other pointer variables that point to elements in the linked list [10]. The complexity of creating linked lists is more complicated than arrays and vectors, but changing the order of elements in linked lists is easier [13].

Seeing the advantages and disadvantages, we will examine how the application of the MinFinder algorithm with large amounts of data, namely 10 ^ 3, 10 ^ 4, and 10 ^ 5 the amount of data. Testing this algorithm is done using C language, for arrays and linked lists. For vectors, testing will be done using the C ++ language. The computer processor used in this test is Intel i7-7200 with 4GB of RAM memory.

## 4. IMPLEMENTATION

The MinFinder pseudocode algorithm previously described, will be implemented in 3 data structures, namely: array, vector, and linked list. Data inputting is done through a file containing a row of random numbers to be sorted. The numbers will be entered into the data structure and sorted using the MinFinder algorithm. After the sorting process has been performed, the output of the data that has been sequentially will be displayed. There are 5 variables used in implementing this MinFinder algorithm, among others.

- minValue: integer variable that holds the smallest value of one iteration in the array array.
- PositionOfMinValue: integer variable that holds the position of minValue.
- L: integer variable that contains the value of the last index in the array that is n-1, with n the amount of data.
- NextIterPoint: Integer variable that functions as a limit starting from iteration, so that the smallest data that has been moved forward, is not compared.
- Finder: label used for repeating markers of commands when called the go to function.

To clarify the function of each variable, you can see the illustration of the MinFinder algorithm in Figure 2.

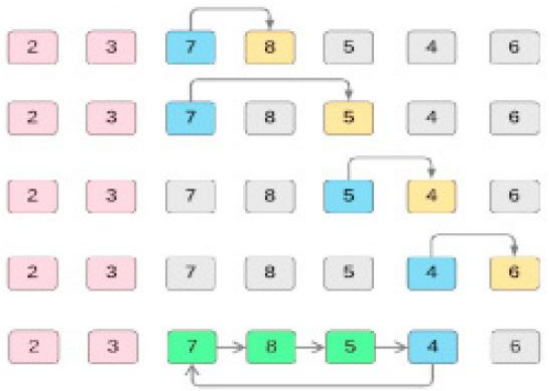

**Figure 2:** Illustration of Iteration on MinFinder

NextIterPoint is marked with a pink box, PositionOfMinValue and minValue are marked with a blue box. minValue will be checked for each element behind NextIterPoint which is marked with an orange box. If an element whose value is smaller than minValue is found, the value of minValue will be updated, and the iteration will continue from the latest PositionOfMinValue.

After the iteration reaches L, it indicates that no more elements need to be checked for the smallest value at that position, and the minValue is moved forward after NextIterPoint. The iteration will then continue from NextIterPoint + 1 to NextIterPoint with the same value as L, which indicates all data has been sorted.

In Figure 4, you can see the MinFinder algorithm consists of 3 iterations. First is the iteration from NextIterPoint to L. This iteration will run as long as the value of i is smaller or equal to L. If the value of i is greater than L, then the iteration will stop, which indicates all data has been sorted. The second iteration can be seen, namely the shift process performed on the element located between NextIterPoint and PositionOfMinValue to the right and move minValue to the NextIterPoint position. Whereas the third iteration, is the goto call which makes the iteration ends and is repeated when there is a value smaller than minValue and i is in the L position

```
while(true) {
    for(..; …; …) {
        if(condition1)
            break;
    }
    if(condition2)
        break;
}
```

**Figure 3:** The goto command if replaced into looping

As shown in Figure 3, it can be seen if the use of goto is equivalent to looping with a combination of breaks. Using too much goto can cause spaghetti code, where the flow of control of the program will be complicated to follow. However, in MinFinder, using goto is appropriate because it is not excessive, and is more practical to use to get out of nested loops.

```
Finder:
minValue = A[positionOfMinValue];
for(i = positionOfMinValue + 1; i<=L; i++)
{
    if (minValue > A[i]) {
        minValue = A[i];
        positionOfMinValue = I;
        if (i != L) goto Finder;
    }
    if (i = L) {
        for(j = positionOfMinValue; j>= NextIterPoint; j--) {
            A[j] = A[j-1];
        }
        A[NextIterPoint] = minValue;
        NextIterPoint++;
        positionOfMinValue = NextIterPoint;
        goto Finder;
    }
}
```

**Figure 4:** Implementation of MinFinder on Array

MinFinder implementation in arrays and vectors does not have significant differences. The difference lies in the use of indexes on dynamic vectors rather than static arrays [12]. The dynamic vector initializer can be seen according to Figure 5.

```
vector<data_type> vector_name;
```

**Figure 5:** Vector Initialization

Inputting data on vectors is done using the push_back() function. This function is similar to the push() function on the stack. In contrast to arrays, push_back() on a vector does not need to define the location of the index, because data will automatically be entered at the very back of the index. The push_back() function can be seen in Figure 6.

```
vector_name.push_back(value);
```

**Figure 6:** Inputting data on the Vector

In Figure 7 shown the significant difference that can be seen is when i is equal to L, which means i has reached the last position on the element to be checked in a loop. When that happens, the data between minValue and NextIterPoint will be shifted to the right, using the insert() function. The begin() function states the position at the first index. minValue will be entered in the begin() + NextIterPoint position. Then, the data element at the begin position + PositionOfMinValue + 1 will be removed from the vector. Because insert() is done first instead of erase (), then the index to be deleted must be added by 1 first.

```
Finder:
minValue = v[positionOfMinValue];
for(i = positionOfMinValue + 1; i<=L; i++)
{
    if (minValue > A[i]) {
        minValue = A[i];
        positionOfMinValue = I;
        if (i != L) goto Finder;
    }
    if (i = L) {
        v.insert(v.begin() + NextInterPoint, minValue);
        v.erase(v.begin() + positionOfMinValue + 1);
        NextIterPoint++;
        positionOfMinValue = NextIterPoint;
        goto Finder;
    }
}
```

**Figure 7:** Implementation of MinFinder on Vector

For implementing MinFinder on a Linked List, it will be easier to use if done on a Double Linked List because it has 2 pointers next and prev which will make it easier to move the minValue to the next NextIterPoint. In a Linked List, struct elements must be defined in a new node to be allocated first, before compiling into a Linked List. Defining a struct data type consists of an integer value, pointer next, and prev.

After defining the struct node in the Linked List and its arrangement, the MinFinder sorting process is then performed. In this Double Linked List, there is a pointer head that points to the front most list, and a tail that points to the rear list. To prevent errors at run time because they point to null, the looping process is limited to n-1 data. So that after the looping process is completed, a comparison will be made once again between the n-1 and n data, as can be seen in Figure 8.

```
Finder:
i = positionOfMinValue -> next;
while(i != NULL && nextIterPoint -> next != tail) {
    if (positionOfMinValue -> value > i -> value) {
        positionOfMinValue = i;
        if (i != tail) goto Finder;
    }
    if (i == tail) {
        if(positionOfMinValue != head &&
        positionOfMinValue != tail &&
        positionOfMinValue != nextIterPoint) {
        i = i -> next;
    } else if (positionOfMinValue == tail &&
            positionOfMinValue != nextIterPoint) {
        nextIterPoint = positionOfMinValue;
        nextIterPoint = nextIterPoint -> next;
        positionOfMinValue = nextIterPoint;
        goto Finder;
    }
}
```

**Figure 8:** Implementation of MinFinder on Linked List

When i is equal to tail, there are 2 conditions if-statements, if PositionOfMinValue is in the middle of the head and tail, and PositionOfMinValue does not refer to NextIterPoint. As can be seen in Figure 9, the conditions for inserting when NextIterPoint are in the head and do not have different treatment. First the next from before PositionOfMinValue is connected to the next from PositionOfMinValue to break the chain. Then prev from next PositionOfMinValue will be associated with prev PositionOfMinValue. The PositionOfMinValue node will be connected to the front of the head if NextIterPoint is still in its initial position, namely in the head. If not, then PositionOfMinValue is associated with the position in front of the head. The same is true if PositionOfMinValue is on the tail.

```
positionOfMinValue -> prev -> next = positionOfMinValue -> next;
if(nextIterPoint == head) {
   positionOfMinValue->next = nextIterPoint;
   positionOfMinValue->prev->next->prev = positionOfMinValue
– prev;
   positionOfMinValue->prev = NULL;
   nextIterPoint->prev = positionOfMinValue;
   head = positionOfMinValue;
} else {
   positionOfMinValue->next = nextIterPoint;
   positionOfMinValue->prev->next->prev =
positionOfMinValue->prev;
   positionOfMinValue->next->prev = positionOfMinValue;
}
```

**Figure 9:** Moving the minValue node to NextIterPoint

## 5. RESULT

After making the code has been completed, the experiment will be carried out. In the trial data that will be tested there are 3 cases, namely data amounting to n = 10 ^ 3, 10 ^ 4, and 10 ^ 5. Each test will be carried out 10 times, then the average time will be searched. The sorting process that occurs will have a different working time speed, which depends on the data structure. The following are the results of the test which can be seen in the Table 1.

**Table 1:** The MinFinder Experiment at n = 10^3

| i-th trial | Array | Vector | Linked List |
|---|---|---|---|
| 1 | 0,4690 | 1,0470 | 0,5470 |
| 2 | 0,4840 | 1,0870 | 0,5310 |
| 3 | 0,5000 | 0,6560 | 0,3910 |
| 4 | 0,5470 | 1,0620 | 0,5160 |
| 5 | 0,5000 | 0,8750 | 0,5160 |
| 6 | 0,4220 | 1,0620 | 0,4840 |
| 7 | 0,2810 | 1,0310 | 0,5940 |
| 8 | 0,5620 | 0,7190 | 0,2500 |
| 9 | 0,5160 | 0,7190 | 0,5160 |

|  |  |  |  |
|---|---|---|---|
| 10 | 0,5160 | 1,0470 | 0,5160 |
| Average | 0,4797 | 0,9035 | 0,4861 |

In experiments using data n = 10 ^ 3, it can be seen that the use of arrays and linked lists is 1.8X faster than using vectors. Linked lists and arrays have a fairly thin difference of 0.0064 seconds, where there are almost no significant differences. The use of arrays is fairly fast and practically superior here.

**Table 2:** The MinFinder Experiment at n = 10^4

| **i-th trial** | **Array** | **Vector** | **Linked List** |
|---|---|---|---|
| 1 | 1,2190 | 2,4100 | 1,3060 |
| 2 | 1,2310 | 2,5000 | 1,3320 |
| 3 | 1,2420 | 2,4120 | 1,3600 |
| 4 | 1,2790 | 2,4370 | 1,3610 |
| 5 | 1,2420 | 4,8600 | 1,2970 |
| 6 | 1,2180 | 2,4150 | 1,3150 |
| 7 | 1,1920 | 2,3850 | 1,2680 |
| 8 | 1,2680 | 2,4670 | 1,2440 |
| 9 | 1,2920 | 2,4610 | 1,2970 |
| 10 | 1,2240 | 2,4620 | 1,2860 |
| Average | 1,2407 | 2,6809 | 1,3066 |

As can be seen in Table 2, in the second experiment using data n = 10 ^ 4, it can be seen that the use of arrays still outperforms this test. Vector is still the test with the longest time which takes 2X longer than arrays and linked lists. Array and linked list have a little time difference too, which is equal to 0.659 seconds

**Table 3:** The MinFinder Experiment at n = 10^5

| **i-th trial** | **Array** | **Vector** | **Linked List** |
|---|---|---|---|
| 1 | 25,1220 | 39,8810 | 40,4350 |
| 2 | 24,9720 | 39,8250 | 40,4320 |
| 3 | 24,8530 | 47,3150 | 40,0600 |
| 4 | 24,9360 | 39,3870 | 40,6060 |
| 5 | 25,0620 | 38,9320 | 42,1510 |
| 6 | 25,1250 | 42,9710 | 43,2060 |
| 7 | 24,9570 | 39,0970 | 42,6490 |
| 8 | 24,9520 | 40,4180 | 45,1760 |
| 9 | 25,2140 | 39,0900 | 39,9970 |
| 10 | 25,0910 | 39,4910 | 40,0530 |
| Average | 25,0284 | 40,6407 | 41,4765 |

In experiments using data n = 10 ^ 5 that can be seen in Table 3, there are differences in results from before. Linked List is the longest of the three data structures. Arrays remain in the fastest position to complete sorting. The difference in speed by using a 1.6X array is faster than vector and linked lists. The difference between vector and linked list, which is equal to 0.8358 seconds.

## 6. CONCLUSION

Based on testing that has been done, it can be concluded that the array with amount of data less than or equal to 10 power of 5 is the best data structure. Arrays require an average of 1.8X faster than vectors and linked lists in the application of the MinFinder algorithm. Vector and linked list can be used when wanting to create dynamic arrays, but if the data gets bigger the performance of both will decrease due to slower data access, than arrays that can directly access their indexes.

## ACKNOWLEDGEMENT

Thank you to the Universitas Multimedia Nusantara, Indonesia which has become a place for researchers to develop this journal research. Hopefully, this research can make a major contribution to the advancement of technology in Indonesia.

## REFERENCES

1. M. Shabaz and A. Kumar, “SA sorting: A novel sorting technique for large-scale data,” Journal of Computer Networks and Communications, vol. 2019, 2019.
2. B. Subbarayudu, L. Lalitha Gayatri, P. Sai Nidhi, P. Ramesh, R. Gangadhar Reddy, and C. Kishor Kumar Reddy, “Comparative analysis on sorting and searching algorithms,” International Journal of Civil Engineering and Technology, vol. 8, no. 8, pp. 955–978, 2017.
3. M. J. Mundra and B. L. Pal, “Minimizing Execution Time of Bubble Sort Algorithm,” vol. 4, no. 9, pp. 173–181, 2015.
4. W. I. Kevin Hendy, “Efficiency Analysis of Binary Search and Quadratic Search in Big and Small Data,” COMPUTATIONAL SCIENCE AND TECHNIQUES, vol. 7, no. 1, pp. 605–615, 2020.
5. M. S. Rana, M. A. Hossin, S. M. H. Mahmud, H. Jahan, A. K. M. Z. Satter, and T. Bhuiyan, “MinFinder: A New Approach in Sorting Algorithm,” Procedia Computer Science, vol. 154, pp. 130–136, 2018.
6. B. K. Joshi, Data Structures and Algorithms in C. New Delhi: Tata Mcgraw Hill Education Private Limited, 2010.
7. V. Andiyani and W. Istiono, “Analysis of Fibonacci Numbers Calculations Using Static Programming and Dynamic Programming Algorithms to Get Optimal Time Efficiency,” International Journal of Open Information Technologies, vol. 8, no. 12, pp. 19–22, 2020.
8. B. Harvey, “Algorithms and Data Structures,” Computer Science Logo Style, vol. 1, no. August 2004, p. 212, 2019.

9. F. Franek, “Memory as a Programming Concept in C and C++,” Memory as a Programming Concept in C and C++, p. 12, 2004.
10. R. Acevedo-Avila, M. Gonzalez-Mendoza, and A. Garcia-Garcia, “A linked list-based algorithm for blob detection on embedded vision-based sensors,” Sensors (Switzerland), vol. 16, no. 6, 2016.
11. Z. Rustam and N. P. A. A. Ariantari, “Comparison between support vector machine and fuzzy Kernel C-Means as classifiers for intrusion detection system using chi-square feature selection,” AIP Conference Proceedings, vol. 2023, 2018.
12. G. M. Seed, An Introduction to Object-Oriented Programming in C++, vol. 49, no. 0. Springer-Verlag London, 2001.
13. J. Katajainen, “Worst-case-efficient dynamic arrays in practice,” Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 9685, no. 1, pp. 167–183, 2016.