

Quick sort and merge sort performance comparison in the flutter framework

^{1*}Achmad Fahreza Alif Pahlevi; ²Suhartono
^{1,2}UIN Maulana Malik Ibrhaim, Malang, Indonesia
^{*}200605110098@student.uin-malang.ac.id

ABSTRACT

This study provides a comparative analysis between Quick sort and Merge sort sorting algorithms in the context of the Dart programming language environment. The results show that Quick Sort dominates in terms of speed when dealing with small data, while Merge sort excels when data is large. Although Merge sort is more efficient in large data size situations, it requires significant additional memory allocation, whereas Quick sort makes more efficient use of memory space. Therefore, the choice between these two algorithms becomes highly dependent on the size of the data and the terms of use. For sorting data with small sizes (under 400 elements), Quick sort is a better choice in terms of performance. On the other hand, Merge sort is recommended for large data. In addition, when utilizing cache locality becomes a priority, Quick sort becomes a more optimal choice for all data sizes.

Keywords: Quick Sort, Merge Sort, Performance, Comparison, Data

1. INTRODUCTION

Data sequencing is one of the classic problems in the field of informatics and technology. The data sorting process is crucial in CAR Hoare introduced the Partition Exchange Sort method, also called Quick Sort, which increased its effectiveness by setting a considerable distance between the two components. The Quick Sort sorting method can be used in both recursive and non-recursive forms. Suppose there are N elements in descending order, and we have to sort those N elements by $N/2$ times. This means replacing the leftmost element with the rightmost one, and then gradually moving towards the central element. However, you can only do so if you believe the order will go down. [1]. However, it is difficult to determine which algorithm is good for a particular situation because the data to be sorted varies widely. Therefore, it is necessary to conduct a comparative analysis between the two algorithms to determine which algorithm is faster and more efficient in sorting data in Dart. Previous research has been conducted to compare sorting algorithms such as Bubble Sort, Shell Sort, and Quick Sort, as well as Bubble Sort, Merge Sort, and Quick Sort [2]. However, research comparing Quick Sort and Merge Sort on Dart is still limited. Therefore, this study will discuss the comparison between Quick Sort and Merge Sort in Dart [3] to determine which algorithm is better at sorting data.

The purpose of this study is to compare the performance of Quick Sort and Merge Sort on Dart. This research will implement both algorithms in Dart and conduct performance and functional testing. Performance testing will be done by measuring the time it takes by each algorithm to sort arrays of different sizes. Functional testing will be done by ensuring that both algorithms produce correctly sorted arrays. The test results will be analyzed to determine which algorithm is better at sorting the data in Dart.

C.A.R. Hoare introduced the Partition Exchange Sort method, also known as the Quick Sort method. This method establishes a considerable distance between the two components to increase their effectiveness. The Quick Sort sorting method can be used both in recursive and non-recursive forms [4]. Suppose there are N elements in descending order, and we have to sort those N elements by $N/2$ times. This means replacing the leftmost element with the rightmost one, and then gradually moving towards the central element. However, you can only do this if you are sure that the order is down [2]. The Merge Sort method that will be described below is also referred to as Two-Way Merge Sorting. The offspring resulting from the merging of two vectors is used in this approach. The input vector is considered to have N elements, and the vector consisting of a vector is also considered to have N vector fruits. For each pair of vectors, a merger process is carried out so that $N/2$ vectors are obtained consisting of 2 elements each (if N is odd, there will be a vector with 1 element). Then the merger was carried out again so that $N/4$ vectors were obtained. Until a vector is in order, this process continues [2].

Dart is an object-oriented, open-source programming language developed by Google [5]. Dart was first introduced in 2011, and has since been used by Google and other companies to develop web, mobile, and server-side applications [6]. Dart is generally combined with Google's framework, Flutter, for mobile app development [7]. However, this programming language can also be used with other frameworks for various purposes. Dart has a variety of functions, and among them are detecting programming errors in the first place, creating programs faster and using fewer computer resources, as well as having a syntax similar to other programming languages such as Java and C++. Dart also has tools to help developers write, test, and improve code more easily so that it is more accurate [7]. Dart can manage code to be organized and easier to understand, which can support the development of more complex applications [8].

2. RESEARCH METHOD

In this study, Quick Sort and Merge Sort will be compared in Dart. The implementation of both algorithms will be carried out using the Dart programming language. Performance testing will be done by measuring the time it takes by each algorithm to sort arrays of different sizes. Functional testing will be done by ensuring that both algorithms produce correctly sorted arrays. The test results will be analyzed to determine which algorithm is better at sorting arrays in Dart. The conclusion of this study will include which algorithm is better at sorting arrays in Dart and why. The following are the details of this research method:

A. Implementation: After the introduction, the next step is to implement both algorithms in Dart. This implementation should include code to sort arrays using Quick Sort and Merge Sort.

```

List<int> quickSort(List<int> list) {
  if (list.length <= 1) {
    return list;
  }
  int pivot = list[0];
  List<int> left = [];
  List<int> right = [];
  for (int i = 1; i < list.length; i++) {
    if (list[i] < pivot) {
      left.add(list[i]);
    } else {
      right.add(list[i]);
    }
  }
  return [...quickSort(left), pivot, ...quickSort(right)];
}

```

Figure 1. Quick Sort Algorithm in Dart

In Figure 1, the Quick Sort (List<int> list) function is a function used to perform Quick sorting, the result will give a return of the sorted are data value

```

List<int> mergeSort(List<int> list) {
  if (list.length <= 1) {
    return list;
  }
  int middle = list.length ~/ 2;
  List<int> left = mergeSort(list.sublist(0, middle));
  List<int> right = mergeSort(list.sublist(middle));
  return gabung(left, right);
}

List<int> gabung(List<int> left, List<int> right) {
  List<int> result = [];
  while (left.isNotEmpty && right.isNotEmpty) {
    if (left.first <= right.first) {
      result.add(left.removeAt(0));
    } else {
      result.add(right.removeAt(0));
    }
  }
  if (left.isNotEmpty) {
    result.addAll(left);
  }
  if (right.isNotEmpty) {
    result.addAll(right);
  }
  return result;
}

```

Figure 2. Merge Sort Algorithm in Dart

In Figure 2, the Merge Sort (List<int> list) function is a function used to sort the Merge, the result will return the sorted are data value

```

void measurements(List<int> list) {
  Stopwatch stopwatch = Stopwatch().start();
  mergeSort(list);
  print('mergeSort() executed in ${stopwatch.elapsed}');
  stopwatch.reset();
  quickSort(list);
  print('quickSort() executed in ${stopwatch.elapsed}');
}

```

Figure 3. Measurement Algorithm in Dart

In Figure 3, the measurements (List<int> list) function is a function used to calculate the execution time in the dart against the Merge Sort and Quick Sort functions.

B. Testing: After implementation, the next step is to test both algorithms. Testing should include performance testing and functional testing. Performance testing should be performed by measuring the time it takes by each algorithm to sort arrays of different sizes. Functional testing should be done by ensuring that both algorithms produce correctly ordered arrays.

C. Analysis: After the test, the next step is to analyze the test results. The performance test results should be compared to determine which algorithm is faster at sorting arrays of different sizes. The results of the functional tests should be analyzed to ensure that both algorithms produce correctly ordered arrays.

D. Conclusion: After the analysis, the final step is to draw conclusions from this study. The conclusion includes which algorithm is better at sorting arrays in Dart and why.

3. RESULT AND ANALYSIS

The time it takes for both algorithms to sort data of different sizes is documented. Small data sizes between 10 and 500 are first evaluated before large data sizes between 1000 and 1000000 are evaluated. This division is done to observe the actual behavior of the two algorithms in the divided section. Table 1 and Table 2 show the time recorded for small and large data sizes, respectively. While Figure 4 and Figure 5 show a graphical representation of the data in Table 1 and Table 2 respectively:

Table 1. Measure Execution Time with small data

Number of inputs (n)	Quicksort (s)	Merge sort (s)
1000	0.14937	0.14123
2000	0.31736	0.27309
3000	0.47618	0.40654
10000	0.47618	0.40654
20000	3.6413	2.6879
40000	6.6647	2.5012
90000	16.1045	6.6277
1000000	85.0534	46.551

Table 2. Measure Execution Time with big data

Number of Inputs (n)	Quicksort(s)	Merge sort (s)
10	0.000989	0.001383
15	0.001813	0.002013
20	0.002692	0.003340
25	0.002677	0.003564
50	0.005854	0.007322
100	0.010904	0.013661
200	0.024704	0.026567
500	0.069018	0.067672

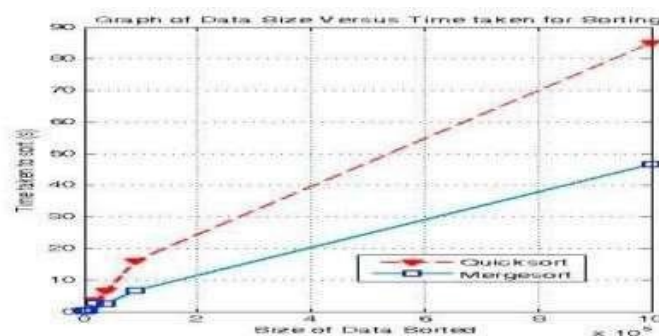


Figure 4. Sorting Time of Quick Sort and Merge Sort ($n \leq 500$)

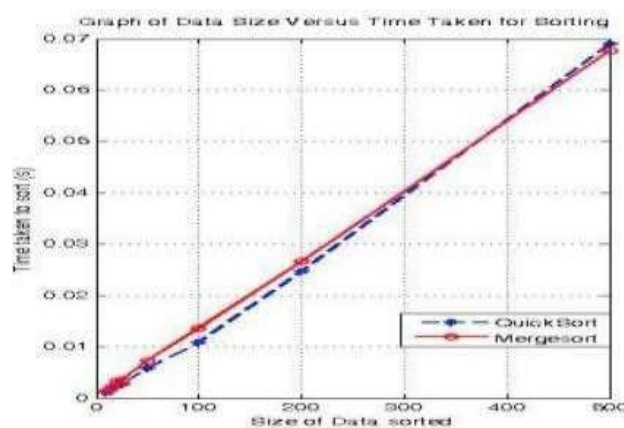


Figure 5. Sorting Time of Quick Sort and Merge Sort ($1000 \leq n \leq 100000$)

The sorting time of the sorting algorithm is fast as shown in the Figure. 4, reveals that less time is required for data sizes less than 400 elements whereas for data sizes of 400 and above, merge sort seems to take less time than Quick Sort. The graph in Figure 5 clearly shows that merge sort is faster than the Quick Sort algorithm in all cases for data sizes in the range of 1000 to 1000000

4. CONCLUSION

The analysis of the two algorithms conducted in the dart programming language environment can be concluded that Quick Sort is faster than merge sort when the data size is small while merge sort is faster when the data size is large. Although merge sort is faster, it requires additional memory space $O(n)$ to store additional arrays while Quick Sort requires $O(\log n)$ space. If there is therefore a need to choose between Quick Sort and Merge sorting algorithms for faster calculations, Quick Sort is preferred over Merge Sort when the data size is small (below 400 elements) while Merge Sort is recommended for large size data. However, where there is a need to take advantage of cache locality, Quick Sort is preferred for all data sizes. It is believed that the information provided in this paper will be invaluable to programmers in choosing when and where to use each of the two algorithms.

REFERENCES

- [1] R. Ranokphanuwat and S. Kittitornkun, "Parallel Partition and Merge Quick Sort (PPMQSort) on Multicore CPUs," *J. Supercomput.*, vol. 72, no. 3, pp. 1063–1091, 2016, doi: <https://doi.org/10.1007/s11227-016-1641-y>.
- [2] I. Ali, H. Nawaz, I. Khan, M. Ameen Chhajro, M. Malook Rind, and A. Maitlo, "Performance comparison between merge and quick sort algorithms in Data Structure," *Int. J. Adv. Comput. Sci. Appl.*, vol. 9, no. 11, pp. 192–195, 2018, doi: <https://doi.org/10.14569/ijacsa.2018.091127>.
- [3] S. Boukhary and E. Colmenares, "A clean approach to flutter development through the flutter clean architecture package," *Proc. - 6th Annu. Conf. Comput. Sci. Comput. Intell. CSCI 2019*, pp. 1115–1120, 2019, doi: <https://doi.org/10.1109/CSCI49370.2019.00211>.
- [4] Z. A. Khalifelu and F. S. Gharehchopogh, "Comparison and evaluation of data mining techniques with algorithmic models in software cost estimation," *Procedia Technol.*, vol. 1, pp. 65–71, 2012, doi: <https://doi.org/10.1016/j.protcy.2012.02.013>.
- [5] J. S. Park, S. N. Jung, M. K. Lee, and J. M. Kim, "Design optimization framework for tiltrotor composite wings considering whirl flutter stability," *Compos. Part B Eng.*, vol. 41, no. 4, pp. 257–267, 2010, doi: <https://doi.org/10.1016/j.compositesb.2010.03.005>.
- [6] S. Y. Ameen and D. Y. Mohammed, "Developing Cross-Platform Library Using Flutter," *Eur. J. Eng. Technol. Res.*, vol. 7, no. 2, pp. 18–21, 2022, doi: <https://doi.org/10.24018/ejeng.2022.7.2.2740>.
- [7] S. Tjandra and G. S. Chandra, "Pemanfaatan Flutter dan Electron Framework pada Aplikasi Inventori dan Pengaturan Pengiriman Barang," *J. Inf. Syst. Hosp. Technol.*, vol. 2, no. 02, pp. 76–81, 2020, doi: <https://doi.org/10.37823/insight.v2i02.109>.
- [8] M. Uciński and M. Dzieńkowski, "A comparative analysis of performance of Flutter and Xamarin development frameworks," *J. Comput. Sci. Inst.*, vol. 25, no. September, pp. 366–370, 2022, doi: <https://doi.org/10.35784/jcsi.3025>.
- [7] R. Ranokphanuwat and S. Kittitornkun, "Parallel Partition and Merge Quick Sort (PPMQSort) on Multicore CPUs," *J. Supercomput.*, vol. 72, no. 3, pp. 1063–1091, 2016, doi: <https://doi.org/10.1007/s11227-016-1641-y>.
- [8] I. Ali, H. Nawaz, I. Khan, M. Ameen Chhajro, M. Malook Rind, and A. Maitlo, "Performance comparison between merge and quick sort algorithms in Data Structure," *Int. J. Adv. Comput. Sci. Appl.*, vol. 9, no. 11, pp. 192–195, 2018, doi: <https://doi.org/10.14569/ijacsa.2018.091127>.
- [9] S. Boukhary and E. Colmenares, "A clean approach to flutter development through the flutter clean architecture package," *Proc. - 6th Annu. Conf. Comput. Sci. Comput. Intell. CSCI 2019*, pp. 1115–1120, 2019, doi: <https://doi.org/10.1109/CSCI49370.2019.00211>.
- [10] Z. A. Khalifelu and F. S. Gharehchopogh, "Comparison and evaluation of data mining techniques with algorithmic models in software cost estimation," *Procedia Technol.*, vol. 1, pp. 65–71, 2012, doi: <https://doi.org/10.1016/j.protcy.2012.02.013>.
- [11] J. S. Park, S. N. Jung, M. K. Lee, and J. M. Kim, "Design optimization framework for tiltrotor composite wings considering whirl flutter stability," *Compos. Part B Eng.*, vol. 41, no. 4, pp. 257–267, 2010, doi: <https://doi.org/10.1016/j.compositesb.2010.03.005>.
- [12] S. Tjandra and G. S. Chandra, "Pemanfaatan Flutter dan Electron Framework pada Aplikasi Inventori dan Pengaturan Pengiriman Barang," *J. Inf. Syst. Hosp. Technol.*, vol. 2, no. 02, pp. 76–81, 2020, doi: <https://doi.org/10.37823/insight.v2i02.109>.
- [13] S. Y. Ameen and D. Y. Mohammed, "Developing Cross-Platform Library Using Flutter," *Eur. J. Eng. Technol. Res.*, vol. 7, no. 2, pp. 18–21, 2022, doi: <https://doi.org/10.24018/ejeng.2022.7.2.2740>.
- [14] S. Tjandra and G. S. Chandra, "Pemanfaatan Flutter dan Electron Framework pada Aplikasi Inventori dan Pengaturan Pengiriman Barang," *J. Inf. Syst. Hosp. Technol.*, vol. 2, no. 02, pp. 76–81, 2020, doi: <https://doi.org/10.37823/insight.v2i02.109>.
- [15] M. Uciński and M. Dzieńkowski, "A comparative analysis of performance of Flutter and Xamarin development frameworks," *J. Comput. Sci. Inst.*, vol. 25, no. September, pp. 366–370, 2022, doi: <https://doi.org/10.35784/jcsi.3025>.