DOKUZ EYLUL UNIVERSITY
ENGINEERING FACULTY
DEPARTMENT OF COMPUTER ENGINEERING

CME 2204 Assignment-I

(Comparison of Merge sort and Quick sort)

By
Bedirhan Karaahmetli

Lecturers
Assoc. Prof. Zerrin IŞIK
Res. Ass. Özlem YERLİKAYA

IZMIR
April, 2023

INTRODUCTION

The management of data is becoming more problematic because of the extraordinary growth of data in the world. The demand for sorting the data is growing as a result. Data sorting algorithms are used because of this. The working logic of two data sorting algorithms, merge sort and quick sort, will be examined in this study, along with some similarities and differences. The research will also discuss the runtime complexity of these algorithms and look at how quickly they can process using arrays as a sample dataset.

SORTING ALGORITHMS

1. MERGE SORT

Working Logic of Merge Sort

Merge sort based on a divide and conquer algorithm. First, it divides the given problem into subproblems. It solves these problems and then combines the solved problems to reach the result.

Merge sort can be easily implemented using recursion. It solves these problems by dividing them into subproblems rather than dealing with the main problem. Then the real problem is solved.

This merge sort algorithm can be explained in 3 steps using arrays:

- ***Divide***: In this step, the array is divided into 2 (or more) parts. This process continues until there are no more pieces to divide. An example of this can be seen in the diagram below.

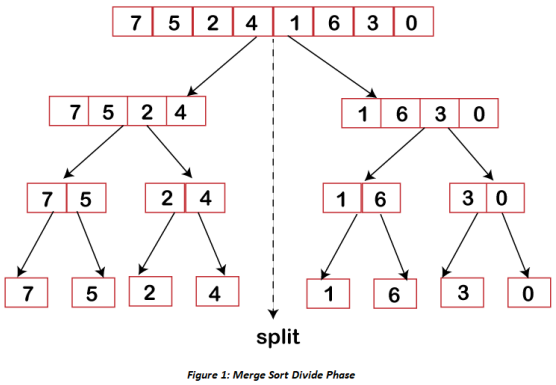


Figure 1: Merge Sort Divide Phase

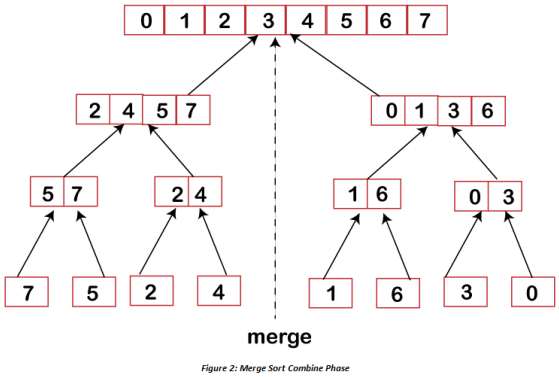
As you can see in the diagram on the left, in the divide phase of the merge sort algorithm, the array is divided into 2 parts until there are no more parts to divide.

Figure 1: Merge Sort Divide Phase

-***Conquer:*** In this step, the divided parts are sorted and combined separately. The merged pieces are then resorted and recombined until the main sequence is completely sorted.

-***Combine:*** In this step, the sorted parts are combined from the bottom to the beginning (from subarrays to the main array).

After the ***Conquer and Combine*** steps, the unsorted array becomes sorted. An example of this process can be seen in the diagram below.



As you can see in the diagram on the left, in the conquer and Combine phase of the merge sort algorithm, the divided parts are sorted and combined separately.

Figure 2: Conquer and Combine Phase

Time Complexity of Merge Sort

Merge sort is a recursive algorithm. So, the time complexity can be expressed as the following recursive relation:

Let T(n) be the total time taken by the *Merge Sort Algorithm*

- Sorting pieces will take the most $2T(\frac{n}{number\ of\ pieces})$ time. Two and three were used as a *number of pieces* in this research. So, it becomes $2T(\frac{n}{2})$ and $2T(\frac{n}{3})$
- Merging the entire array takes O(n) time.

Based on all this, the relational formula will be:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) \quad or \quad T(n) = 2T\left(\frac{n}{3}\right) + O(n)$$

And after solving these formulas, the result will be:

$$O(n) = \log_2 n \quad or \quad O(n) = \log_3 n$$

These results are true for the worst, average, and best cases, as the arrays are divided into parts, sorted, and merged each time.

In conclusion, merge sort makes sense for sorting large data sets because it is efficient and easy to implement.

2. QUICK SORT

Working Logic of Quick Sort

Quick Sort is also a divide and conquer algorithm. It works by selecting a pivot element from the input array, partitioning the remaining elements into two subarrays based on the pivot, and recursively sorting the subarrays.

This quick sort algorithm can be explained in 3 steps using arrays:

- ***Selecting the Pivot Element***: There are different variation of quicksort where the pivot element is selected from different positions. Sample of these are:

  - Selecting the ***leftmost*** element of the array as a pivot element
  - Selecting the ***rightmost*** element of the array as a pivot element
  - Selecting the ***random*** element of the array as a pivot element
  - Selecting the ***median*** value as the pivot elements from the leftmost, middle, and rightmost elements in the array

In this study, the first, second and last ones were applied. But selecting a random element is the same of the selecting leftmost element. Differently, after the selection of a random element, the element is swapped with leftmost element. And it becomes selection the leftmost element.

- ***Partitioning***: The key process in quick sort is a partition. The target of partitions is to put the pivot in its correct positions.

After partitioning the smaller elements from pivot element will be on the pivot element's left, the bigger elements from pivot element will be on the pivot element's right. This phase will be done in linear time.

This process will continue recursively for the sorting the subarrays using quick sort.

- ***Sorting***: Actually, this is not a certain step. The array will be sorted after the recursion. When partitioning the subarrays recursively all elements will be the right place.

An example of this process using pivot as a leftmost element of an array can be seen in the diagram below:

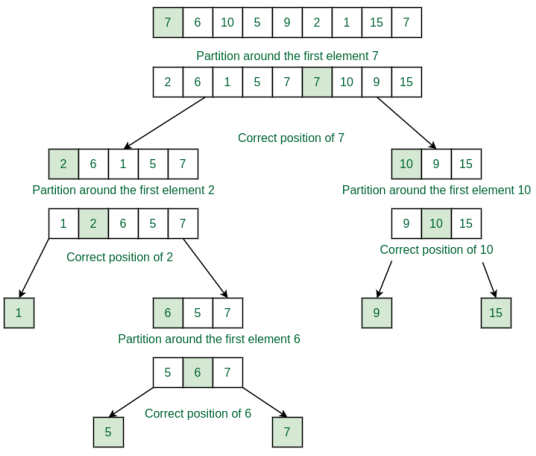


Figure 3: Example of quick sort

Basically, explanation of this algorithm:
- Initialize index of right most element k = high
- Traverse from i = high to low
- If array's i th element greater than pivot:
- Swap array's i th element and k th element
- Decrease k
- At the end swap array's leftmost element and k th element.
- By doing these recursively the array will be sorted at the end.

Time Complexity of Quick Sort

Quick sort is also a recursive algorithm. So, the time complexity can be expressed as the following recursive relation:

Let T(n) be the total time taken by the *Quick Sort Algorithm. At each step, the input of size N is divided into two parts j and (n-j).* The relational formula will be:

$$T(n) = 2T(j) + T(n-j) + M(n)$$

where

- T(n) = Time complexity of Quick Sort for input of size n.
- T(j) = Time complexity of Quick Sort for input of size j.
- T(n-j) = Time complexity of Quick Sort for input of size (n-j).
- T(n) = Time complexity of finding the pivot element for N elements.

1- **Best Case Complexity**:
When the partitioning algorithm always chooses the middle element or near the middle element as the pivot, the best scenario happens. And time complexity will be:

$$O(n) = n \log n$$

2- **Average Case Complexity**:
When the array elements are in a disordered sequence that isn't increasing or decreasing properly, the average scenario happens. And time complexity will be:

$$O(n) = n \log n$$

3- **Worst Case Complexity**:
Worst scenario will happen when selecting the leftmost or rightmost element as a pivot. And time complexity will be

$$O(n) = n^2$$

In conclusion, quick sort makes sense for sorting large data sets when pivot is chosen correctly. It has average time complexity of $O(n) = n \log n$.

SIMILARITIES AND DIFFERENCES

Similarities:
- Both has a divide and conquer algorithms.
- Both have an average time complexity of $O(n) = n \log n$
- Both can be used to sort large datasets efficiently.

Differences:
- Merge Sort is stable, while Quick Sort is not stable.
- Merge Sort has a worst-case time complexity of $O(n) = n \log n$, while Quick Sort has a worst-case time complexity of $O(n) = n^2$
- Merge Sort requires additional memory for merging the sorted subarrays, while Quick Sort does not.

CONCLUSION

In conclusion, Merge Sort and Quick Sort are two popular sorting algorithms that have their unique features and advantages. Merge Sort is suitable for large datasets that require stable sorting, while Quick Sort is ideal for large datasets that require efficient sorting.

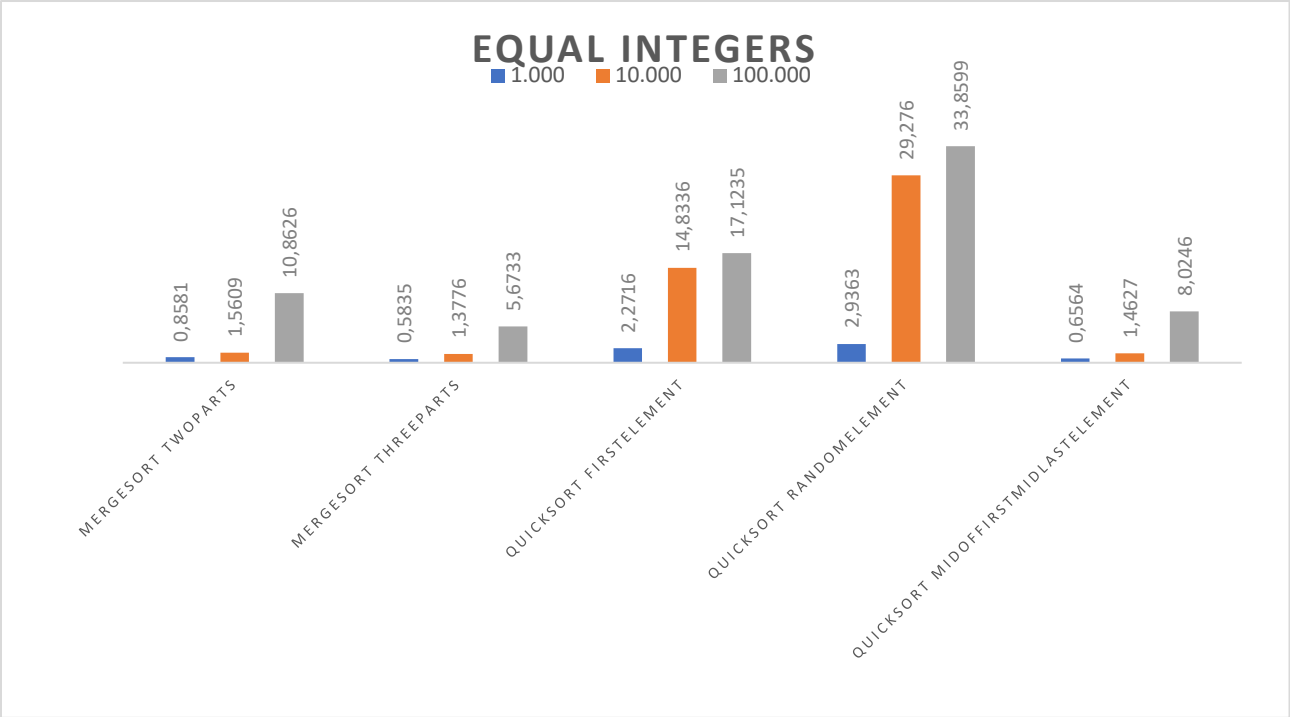
RUNTIME ANALYSIS COMPARISON OF MERGE AND QUICK SORT

In this study, 2 different merge sort methods and 3 different quick sort methods were tested by using 4 different array types and 3 different sizes of data from each array type. You can examine the observed results from the table and graphs below.

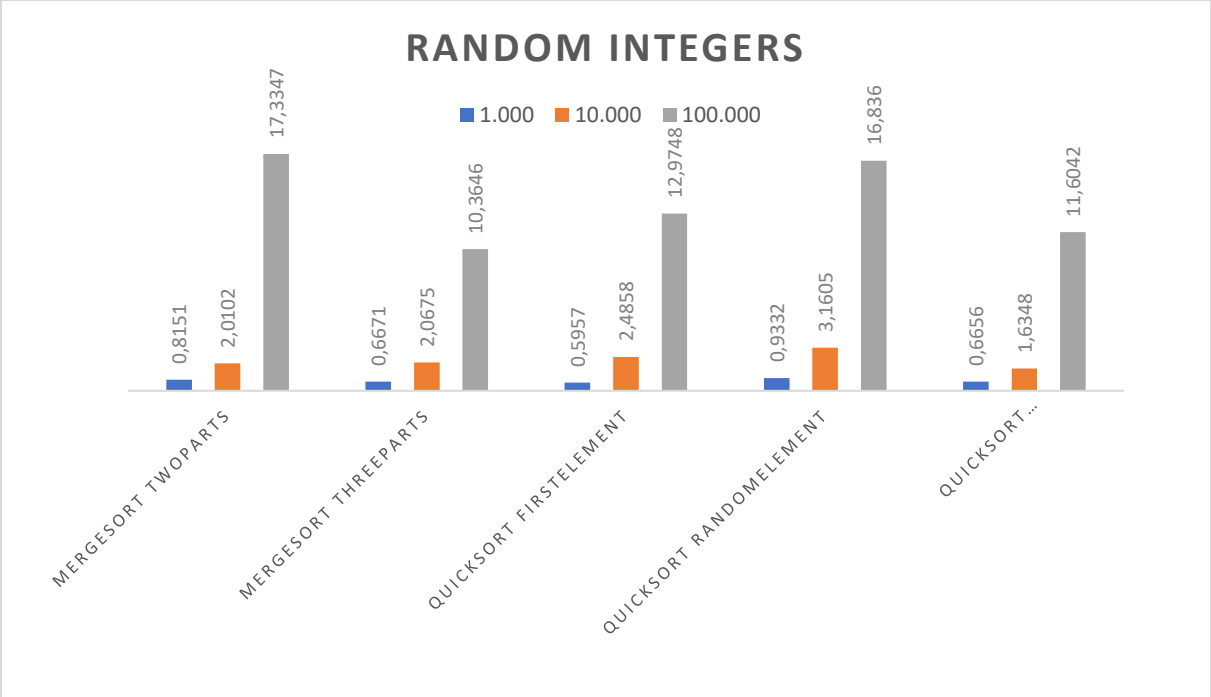
PS: The results obtained varied according to the processor type and the density of the processor, and each was repeated many times for the optimum result, and average results were tried to be obtained.

	EQUAL INTEGERS			RANDOM INTEGERS		
	1.000	10.000	100.000	1.000	10.000	100.000
mergeSort TwoParts	0,8581	1,5609	10,8626	0,8151	2,0102	17,3347
mergeSort ThreeParts	0,5835	1,3776	5,6733	0,6671	2,0675	10,3646
quickSort FirstElement	2,2716	14,8336	17,1235	0,5957	2,4858	12,9748
quickSort RandomElement	2,9363	29,276	33,8599	0,9332	3,1605	16,836
quickSort MidOfFirstMidLastElement	0,6564	1,4627	8,0246	0,6656	1,6348	11,6042

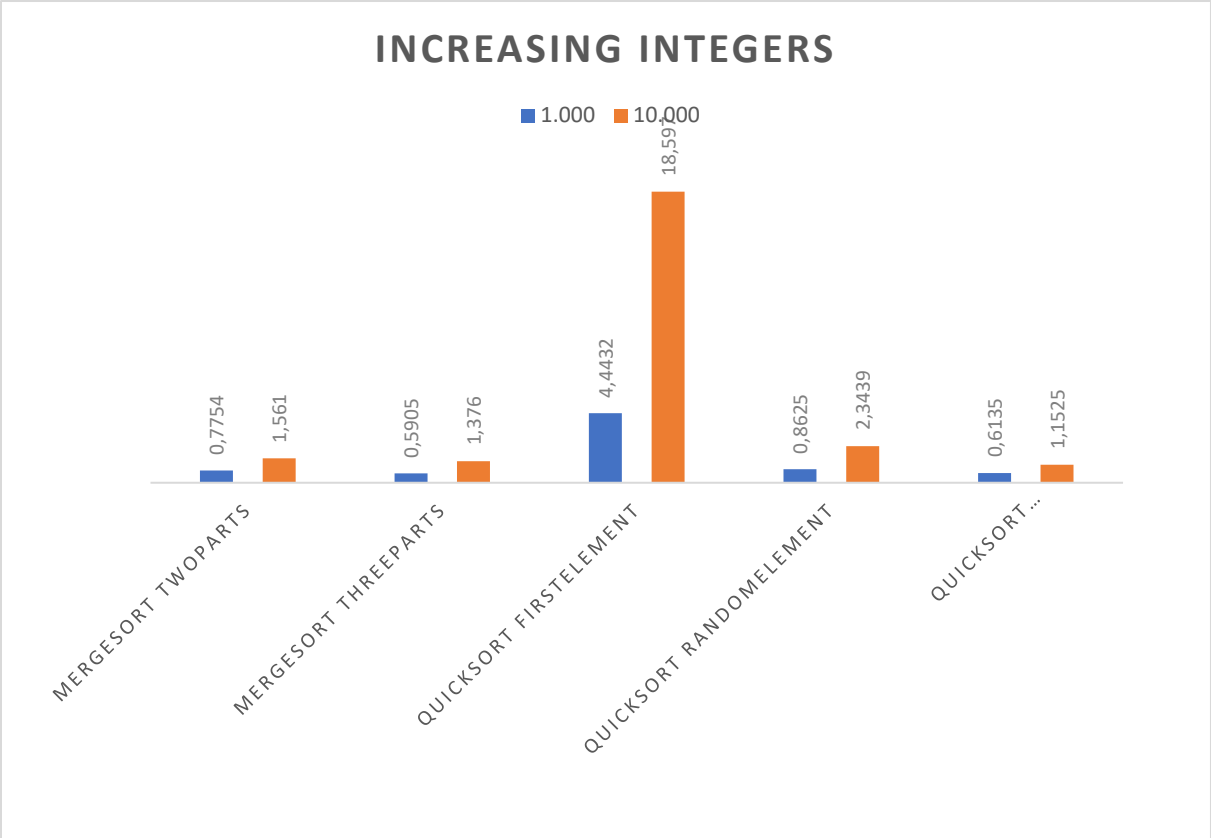
	INCREASING INTEGERS			DECREASING INTEGERS		
	1.000	10.000	100.000	1.000	10.000	100.000
mergeSort TwoParts	0,7754	1,561	11,1179	0,7968	1,5878	10,552
mergeSort ThreeParts	0,5905	1,376	6,1204	0,5956	1,3332	6,5628
quickSort FirstElement	4,4432	18,597	927,9757	3,6346	28,9338	2464,9
quickSort RandomElement	0,8625	2,3439	11,9811	0,9507	2,3292	11,551
quickSort MidOfFirstMidLastElement	0,6135	1,1525	12,4647	0,5779	1,3204	8,1084



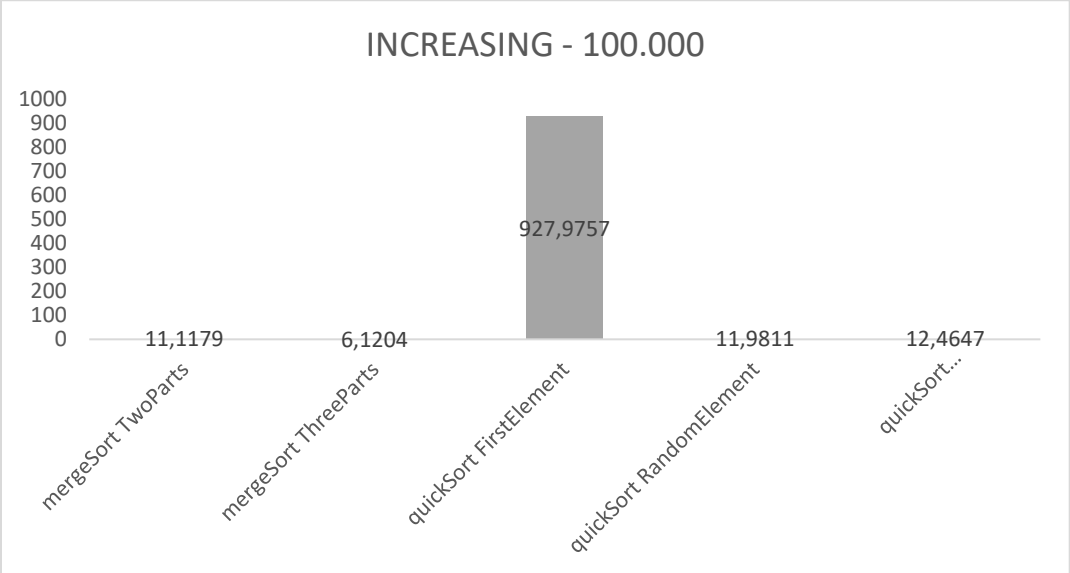
For equal integer array, generally merge sort is faster than quick sort.
For merge sorts, it seems more efficient to divide the array into three parts.
For quick sorts, choosing the pivot as the median is the most efficient method, while choosing the pivot randomly is observed as the most inefficient method.



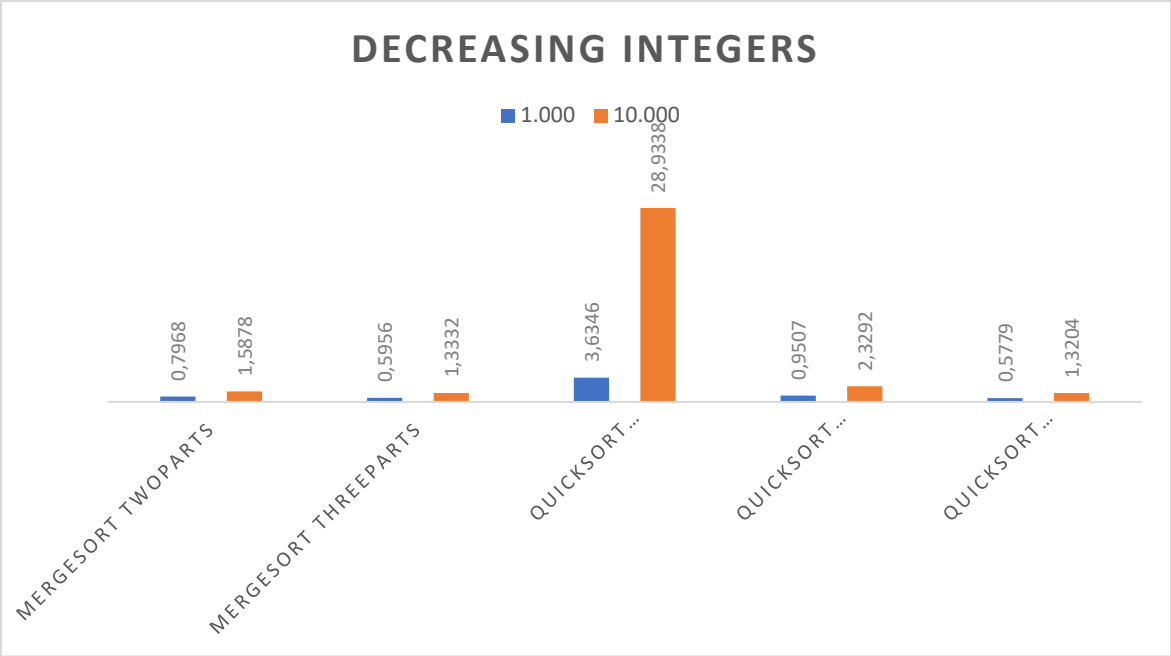
For random integer array, generally merge sort is faster than quick sort with small datasets rather than the larger datasets.
For merge sorts, it seems also more efficient to divide the array into three parts.
For quick sorts, choosing the pivot as the median is also the most efficient method, while choosing the pivot randomly is observed as the most inefficient method.



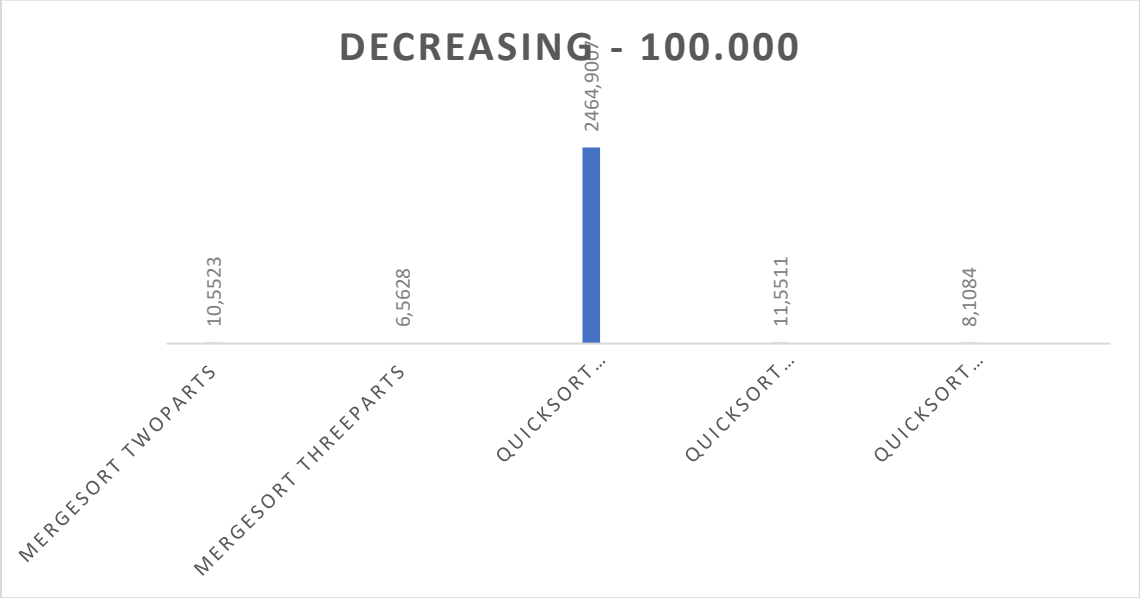
For increasing integer array with small data, merge and quick sort is very close to each other except quick sort with selecting the pivot as a first element.
For merge sorts, it seems slightly better divide the array into three parts.
For quick sorts, choosing the pivot as the median is slightly better than choosing the pivot as a random element.



But if data set gets bigger, the quick sort with choosing first element as a pivot is very slow.



For decreasing integer array with small data, merge and quick sort is very close to each other except quick sort with selecting the pivot as a first element.
For merge sorts, it seems slightly better divide the array into three parts.
For quick sorts, choosing the pivot as the median is slightly better than choosing the pivot as a random element.



But if data set gets bigger, the quick sort with choosing first element as a pivot is also very slow.

PART 3 – SAMPLE SCENARIOS

There are two scenarios that should be specified in the report and the part where the sorting algorithms that make the most efficient for them are explained.

-***First One***:

You have a Turkish-English dictionary with a single word for each word in alphabetical order and you want to translate it into an English-Turkish dictionary. For example; if your Tur-Eng dictionary contains [ayı : bear, bardak : glass, elma : apple, kitap : book] then your Eng-Tur dictionary should be [apple : elma, bear : ayı, book : kitap, glass : bardak]. If we think that there are thousands of words in your dictionary, which sorting algorithm do you use to do this translation faster?

- When translating between Turkish - English dictionaries, we can think of them as random data sets, since there is no specific order. And since these data sets will be large, the merge sort algorithm we use requires more memory usage, so the quick sort algorithm will work more efficiently.

But since we want speed rather than efficiency, the merge sort algorithm will be faster than the quick sort algorithm and the insertion sort algorithm we learned before to solve this problem.

Therefore, I would prefer to use the quick sort algorithm.

-***Second one***:

When you inquire Sub-Upper Pedigree, an ordered list of people according to their birth date comes out in the system of e-Devlet. However, you want to rank the people from the youngest to the oldest one. If you are asked to do this operation using a sorting algorithm, which algorithm do you use?

- For this problem, if I need to use one of the sorting algorithms used in the study, it will be more efficient to use the merge sort algorithm because people come according to their age and the data set will be small.

But when it comes to bottom-upper-pedigree, the first thing that comes to mind is the heap. Heaps are divided into minimum and maximum. In this case, since we are asked to sort from young to old, the minimum heap sort algorithm will be the most appropriate solution.

Therefore, my preference would be the minimum heap sort algorithm.

REFERENCES

Figure 1 - https://www.javatpoint.com/daa-merge-sort

Figure 2 - https://www.javatpoint.com/daa-merge-sort

Figure 3 - https://www.geeksforgeeks.org/implement-quicksort-with-first-element-as-pivot/

Merge Sort

- https://www.baeldung.com/java-merge-sort

- https://www.javatpoint.com/daa-merge-sort

- https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort

- https://www.scaler.com/topics/merge-sort-time-complexity/

Quick Sort

https://www.geeksforgeeks.org/implement-quicksort-with-first-element-as-pivot/

- https://www.geeksforgeeks.org/quick-sort/

- https://www.javatpoint.com/daa-quick-sort

- https://www.programiz.com/dsa/quick-sort

- https://towardsdatascience.com/an-overview-of-quicksort-algorithm-b9144e314a72

- https://iq.opengenus.org/time-and-space-complexity-of-quick-sort/

https://www.interviewkickstart.com/learn/merge-sort-vs-quicksort-performance-analysis