Dokuz Eylul University
Engineering Faculty
Department of Computer Engineering

COMPARISON OF MERGE SORT AND QUICK SORT

By Vedat KARA 2020510046

Lecturers:
Assoc. Prof. Zerrin ISIK
Res. Asst. Ali Cuvitoglu
Res. Asst. Ozlem Yerlikaya

Introduction

This report aims to compare and analyze two divide and conquer algorithms: Merge-sort and quick-sort. A divide and conquer algorithm is a strategy of solving a large problem by dividing that large problem into smaller sub-problems. After that solving them individually and lastly combining them to get the wanted result. To sum up, divide and conquer algorithms have 3 parts: Divide, conquer and combine.

How Merge-Sort Works?

Merge-Sort works by dividing the given array to the smaller ones, sorting each array individually and merging the smaller arrays back. This way we end up with a sorted version of the original array. As the other divide and conquer algorithms merge-sort uses recursive method in order to do these operations.

In this assignment particularly, we have given a mergeSort() function which takes an integer array and number of partitions as parameters.

2-Way Merge Sort: Divides the array into two parts in each recursive level. Firstly, initialize the two new arrays, left and right, and fill them with elements. Then compares each array's indexes one by one (only by increasing the index of the array which has smaller element) and add smaller one first to the original array. After that if there is a remaining element of either of the arrays add them to the end of the original array.

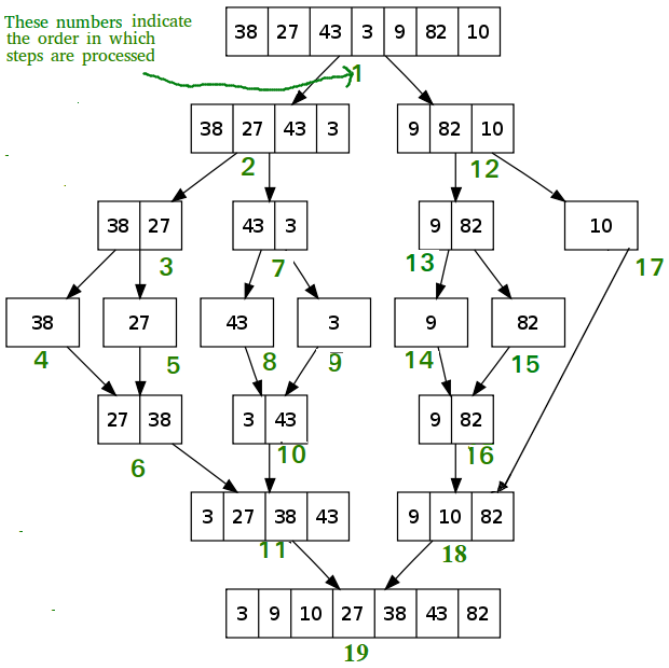


Figure1: Diagram of 2-Way Merge-Sort

3-Way Merge Sort: Divides the array into three parts in each recursive level. The sorting and merging process is the same with 2-way, but there are 3 arrays in this algorithm.

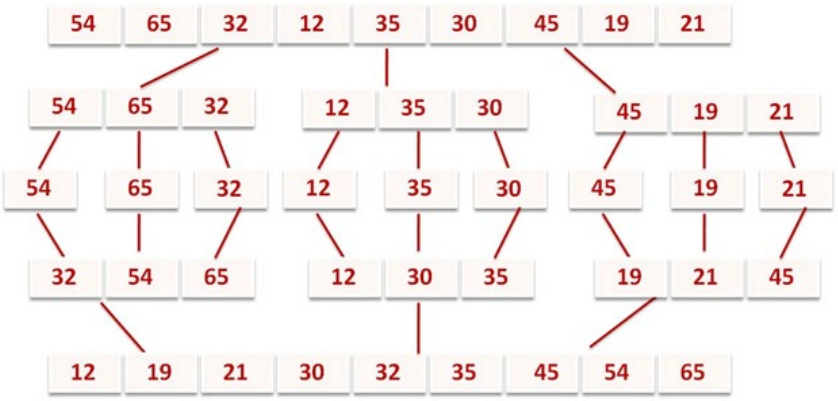


Figure 2: Diagram of 3-Way Merge-Sort

How Quick-Sort Works?

Quick-Sort uses pivot to divide the array that wanted to be sorted. There are different ways of selecting the pivot. Once pivot is selected, reindex the pivot to the correct position where elements at left of the pivot are smaller than pivot and elements at right of the pivot are bigger than pivot. After that array partition occurs. Then each sub-array repeats previous steps until no further division can be done. Quick-sort also uses a recursive method.

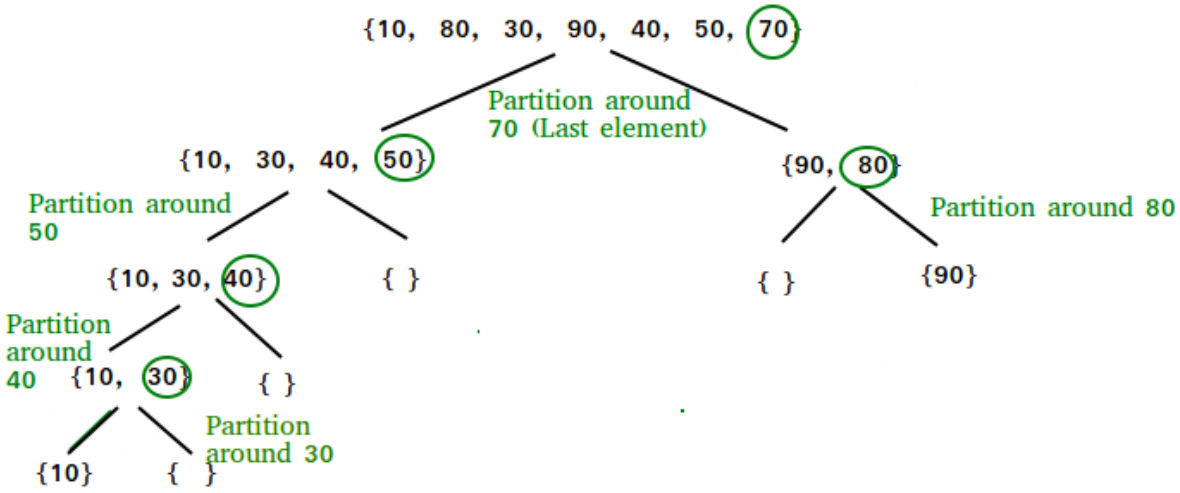


Figure 3: Diagram of Quick-Sort

In this assignment particularly, we have given a quickSort() function which takes an integer array and pivot type as parameters.

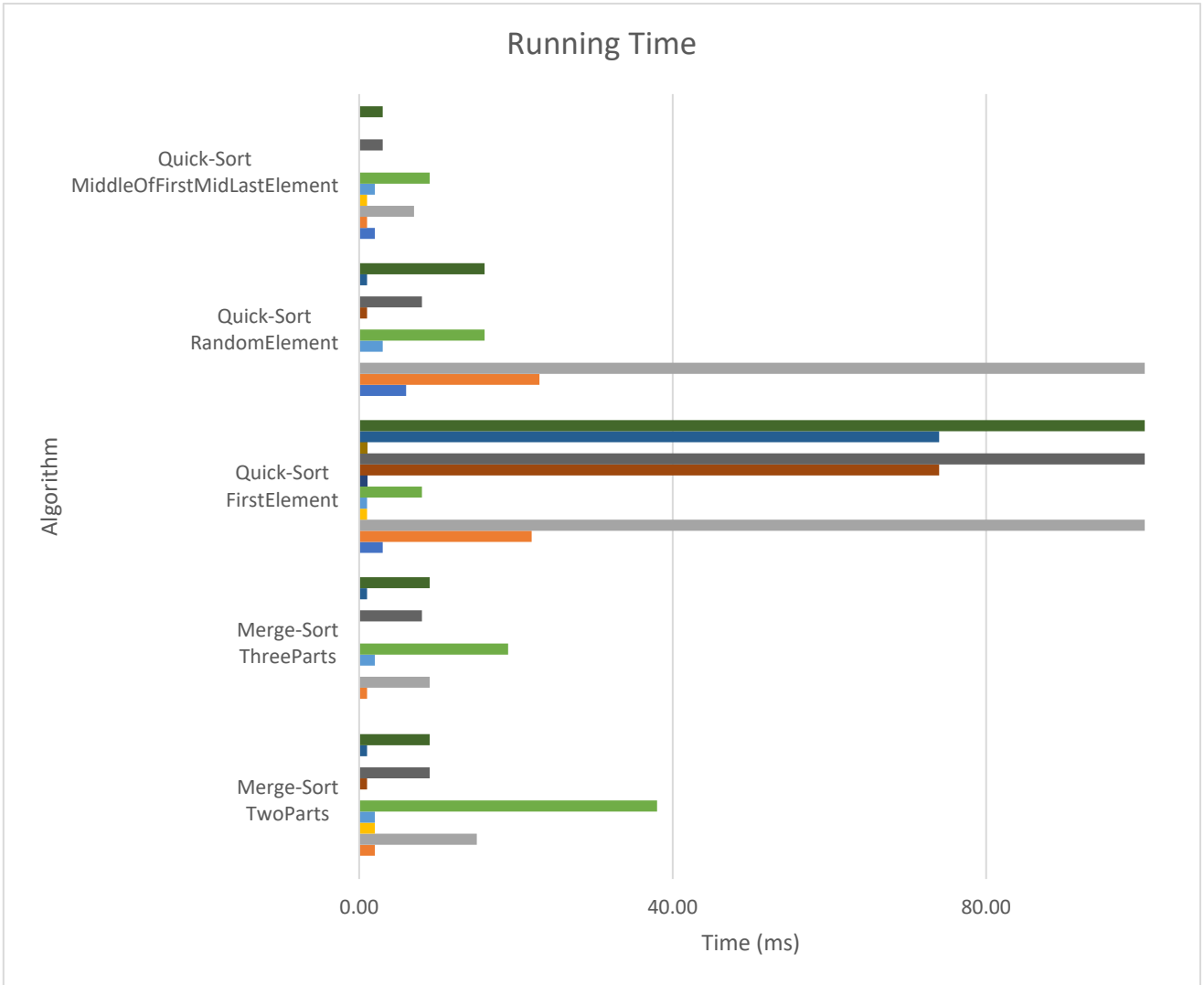
First Element: Selects the first element(left most) as pivot. Then starts from the last element(right-most) and scans the array until reaches to pivot. While scanning, compares the elements with pivot and swaps bigger ones to right of the array. When the scan has completed, put pivot into its correct position. Before the division, array has an order that elements at pivot's right are bigger than pivot and elements at the pivot's left smaller than pivot. Next, divide happens, and the process continues recursively until no further division can be done.

Random Element: Selects a random element as pivot and puts it at the end of the array. Then starts from the first element and scans the array until reaches pivot. While scanning, compares the elements with pivot and swaps smaller ones to left of the array. When the scan has completed, puts pivot into its correct position. Before the division, array has an order that elements at pivot's right are bigger than pivot and elements at the pivot's left smaller than pivot. Next, divide happens, and the process continues recursively until no further division can be done.

Median of 3: If the array size is bigger than 3, find median of three elements: first, middle and last. After that rearrange elements and move the median next to last element. Begin the scanner pointers from first element and the element right before the last element(pivot). We do this because we already know that they are sorted and they are in the right places. Lastly do the usual partition process.

Running Time Table

	EQUAL INTEGERS			RANDOM INTEGERS			INCREASING INTEGERS			DECREASING INTEGERS		
	1,000	10,000	100,000	1,000	10,000	100,000	1,000	10,000	100,000	1,000	10,000	100,000
Merge-Sort TwoParts	0ms	2ms	15ms	2ms	2ms	38ms	0ms	1ms	9ms	0ms	1ms	9ms
Merge-Sort ThreeParts	0ms	1ms	9ms	0ms	2ms	19ms	0ms	0ms	8ms	0ms	1ms	9ms
Quick-Sort FirstElement	3ms	22ms	1943ms	1ms	1ms	8ms	1ms	74ms	7110ms	1ms	74ms	7085ms
Quick-Sort RandomElement	6ms	23ms	1271ms	0ms	3ms	16ms	0ms	1ms	8ms	0ms	1ms	16ms
Quick-Sort MiddleOfFirstMidLastElement	2ms	1ms	7ms	1ms	2ms	9ms	0ms	0ms	3ms	0ms	0ms	3ms



Time Complexity Analysis

Merge-Sort

The time complexity of merge-sort is the same with all 3 cases: best, average and worst. $\Theta(n\log n)$ is asymptotic running time complexity of merge-sort.

2-Way Merge Sort: $\Theta(n\log_2 n)$

$T(n) = 2T(n/2) + \theta(n)$

$a = 2,\ b = 2, f(n) = n$ $\quad n\hat{}\log_2 2 = n?$

$= \log_2 2 = 1$

$= n\hat{}\log_2 2 = n$ case 2

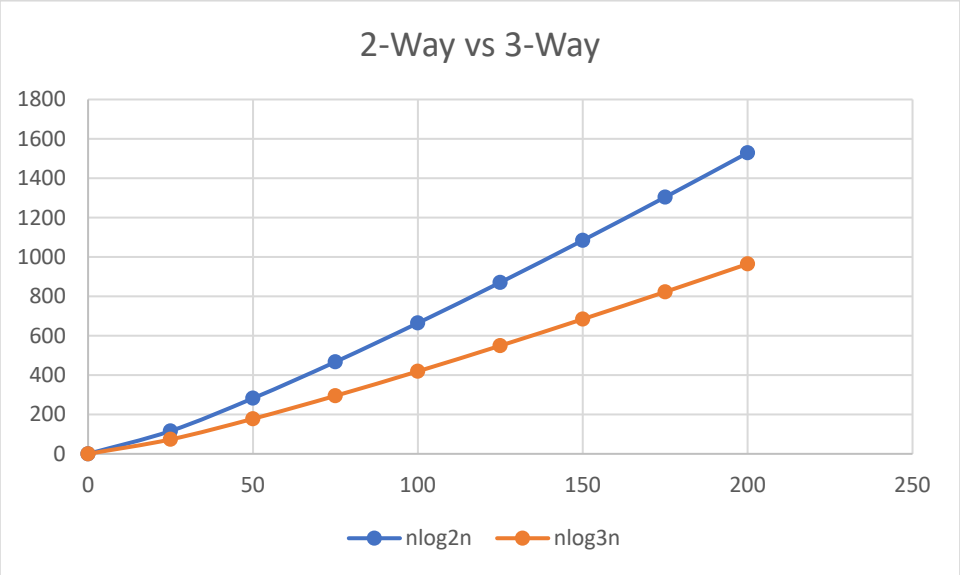
3-Way Merge Sort: $\Theta(n\log_3 n)$

$T(n) = 3T(n/3) + \theta(n)$

$a = 3,\ b = 3, f(n) = n$ $\quad n\hat{}\log_3 3 = n?$

$= \log_3 3 = 1$

$= n\hat{}\log_3 3 = n$ case 2



Quick-Sort

The time passed by quick sort depends on size of the array and partition strategy. Theoretically,

Worst Case: The worst-case is that pivot is the smallest or biggest element in the array we want to be sorted and running time complexity is $\Theta(n^2)$.

First Element or Last Element: If array have elements in increasing or decreasing order.

Random Element: If array has equal elements.

$$T(n) = T(n-1) + \Theta(n)$$

Guess: $T(n) = n^2$

Substitution:

Upper Bound $T(n) <= n^2 + cn$, if $cn >= 0$,
Lower Bound $T(n) >= n^2 + cn$, if $cn <= 0$.

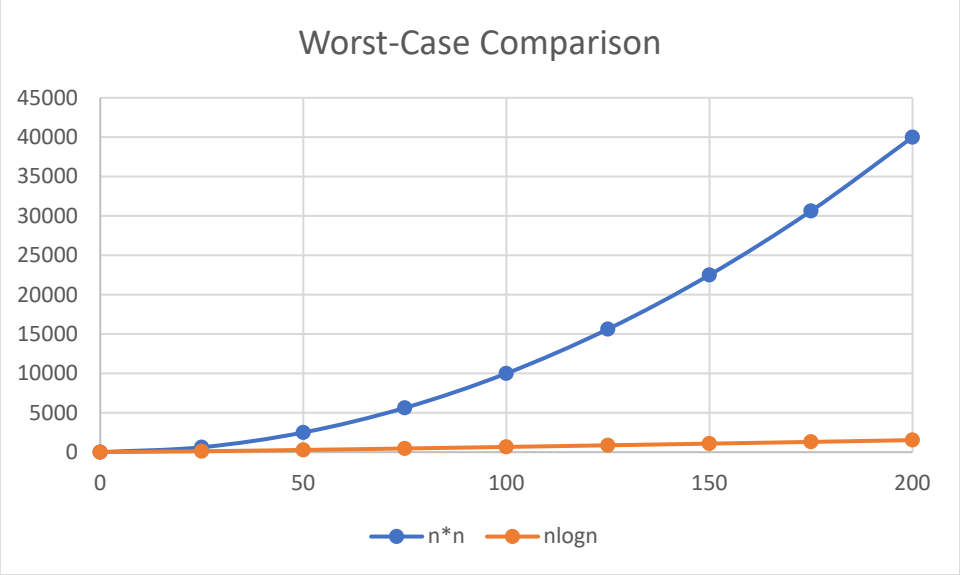
Average Case: The average-case running time complexity is Θ(nlogn).

Best Case: The best-case is that pivot is always the middle element of the array we want to be sorted and running time complexity is Θ(nlogn).

$$T(n) = 2T(n/2) + \Theta(n)$$

$a = 2,\ b = 2, f(n) = n$ $\quad n^{\log_2 2} = n?$

$= \log_2 2 = 1$
$= n^{\log_2 2} = n$ case 2



Considering ‘running time’ table,

In general, merge sort is better than quick sort when data portion is small. Quick sort is not so efficient on very small partitions. Moreover, when the array contains equal, increasing or decreasing integers (worst case scenarios for quick-sort) merge sort is faster without a doubt. However, quick sort is way more efficient when it comes to big data sets (millions, billions…), especially with median of 3 partition strategy.

In the theoretical part we can see that 3-way merge sort has better time complexity than 2-way, we can prove that by looking at the table. We can see the difference clearer when array size is larger. We can also observe that worst, average, and best cases have the same running time complexity.

Quick sort with first element partition strategy is so slow on arrays containing equal, increasing and decreasing elements. Relatively choosing to pivot randomly is more efficient than first element strategy. But the quick sort with median of 3 partition strategy gives the perfect performance on each situation.

To sum up, if you are dealing with a small data set, using merge sort algorithm would be more efficient. On the other hand, quick-sort with median of 3 partition strategy has noticeable performance and efficiency at sorting big data sets.

Questions

Question: 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?

Answer: Between merge-sort and quick-sort algorithms using 3-way merge-sort algorithm would be more efficient to do this translation. Because, thousand of words are represents a small data set and as we discussed previously, merge-sort is best for small data sets. Turkish words are in an order but their equivalents in English is random. As we can see from the table 3-way merge sort is the fastest.

Question: 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?

Answer: Between merge-sort and quick-sort algorithms, using quick sort algorithm with median of 3 partition strategy is better to do this operation. The reason is, we are talking about a data set with decreasing order and we want to reverse it. As we can see on the running time table quick sort with medOf3 is the quickest on this operation particularly. We can not choose first element partition strategy because it is the worst case scenario and random element partition is a gamble.

If we could include heap sort and tree sort, min-heap sort would be more efficient. Because the data set is small (Sub-Upper Pedigree goes to only 1880's), time complexity is the same with merge-sort and quick-sort but heap sort use less space and easier to implement beacuse it does not contain recursive method.

References

- https://www.geeksforgeeks.org/merge-sort/
- https://www.geeksforgeeks.org/quick-sort/
- https://www.geeksforgeeks.org/implement-quicksort-with-first-element-as-pivot/
- https://www.geeksforgeeks.org/quicksort-using-random-pivoting/
- https://github.com/taky2/QuickSort-Median-of-3/blob/master/QuicksortMedian3.java
- https://www.geeksforgeeks.org/heap-sort/
- https://www.geeksforgeeks.org/tree-sort/
- Figure 1 and 3: https://www.geeksforgeeks.org/
- Figure 2: https://www.mycareerwise.com/

Note: Calculation images of time complexity notations created by me.