CME 2204 Assignment-1 Report

Vm options: " -Xss26m " *Table1*

Millisecond (10^-3sec)	**Equal Integers**			**Random Integers**			**Increasing Integers**			**Decreasing Integers**		
Size	**1000**	**10000**	**100000**	**1000**	**10000**	**100000**	**1000**	**10000**	**100000**	**1000**	**10000**	**100000**
mergeSort TwoParts	0.8314	1.255	8.0428	0.3909	1.2679	14.9134	0.0735	0.8937	6.1822	0.0314	0.3028	3.7818
mergeSort ThreeParts	0.4087	1.1789	9.1146	0.4608	1.9137	14.7738	0.0682	0.8732	4.3053	0.0311	0.3727	4.5
quickSort FirstElement	5.879	9.5596	939.9956	0.3126	0.9884	5.6289	0.251	16.6565	1392	0.4308	40.7049	4101.441
quickSort Random	3.6034	16.3803	900.6277	0.4104	1.1475	12.1314	0.0751	0.5024	4.5442	0.052	0.481	5.3468
quickSort MidOfFirst	0.2756	0.4903	1.6337	0.2055	0.8756	6.299	0.0314	0.1713	2.0477	0.015	0.1653	1.881

Comparison of mergeSort types:

Time complexity	**mergeSort TwoParts**	**mergeSort ThreeParts**
Worst Case Time Complexity **[Big-O]**	$\mathbf{O(n * \log_2 n)}$	$\mathbf{O(n * \log_3 n)}$
Best Case Time Complexity **[Big-omega]**	$\mathbf{\Omega(n * \log_2 n)}$	$\mathbf{\Omega(n * \log_3 n)}$
Average Time Complexity **[Big-theta]**	$\mathbf{\Theta\ (n * \log_2 n)}$	$\mathbf{\Theta\ (n * \log_3 n)}$

The only thing that is different is the base of the logarithm in those complexity notations. The base represents the number of sections. In the cases we examine, 2-way mergeSort has 2 as a base in the logarithm and 3-way mergeSort has 3 as a base in the logarithm. A bigger partition (3-way mergeSort in our case) results in lesser recursive call depth. But it does not mean 3-way is faster. There are other parameters that affect our runtime. If we select 3-way merge sort, we have to compare each of 3 sub-array and that comparison (merge) causes more runtime in each recursion step than 2-way mergeSort's recursion steps. On the other hand, 2-way merge sort goes deeper in recursive calls however it has a much lesser merge runtime in each step according to 3-way mergeSort. As we can see in the benchmark table we cannot directly say one is better than the other one. **To Sum Up, in 3-way mergeSort each iteration will have a higher overhead, and hence the total time required will still more or less remain the same compared to 2-way mergeSort's total runtime.**

Comparison of quickSort types:

Time complexity	**quickSort FirstElement**	**quickSort Random**	**quickSort MidOfFirstMidLastElement**
Worst Case Time Complexity **[Big-O]**	O(n^2)	O(n^2)	O(n^2)
Best Case Time Complexity **[Big-omega]**	Ω(n*logn)	Ω(n*logn)	Ω(n*logn)
Average Time Complexity **[Big-theta]**	Θ(n*logn)	Θ(n*logn)	Θ(n*logn)

QuickSort optimization is based on pivot selection. To have the most optimal sort (balanced tree is the most optimal in tree notation) pivot should be as close to the median value as possible. If we formalize time complexity it will be T(n)=T(n/2^x)+O(n) (constants are ignored). X corresponds to the depth of recursion and the x value changes as the pivot's distance to the value of the median changes. The smaller x means the better pivot selection and the lesser recursive calls plus faster partition. But we should not waste much time while finding the optimal pivot because it increases the O(n) in each stage of recursive calls and will be resulted in much more total sort time. So, we can understand that the time spend on O(n) is essential.

By examining “Equal Integers” part of *Table1* we see that MidOfFirstLastElement is the fastest. The reason of **MidOfFirstMidLast** is the fastest even for the big arrays is the pivot’s location which is the middle of the array (if all elements are equal in the array given my algorithm selects the pivot’s index as mid-index of the given sub-array). That type of pivot selection makes branching much more optimal and causes fewer recursive calls. If we show it in tree notation, the tree will be balanced and sorting will be much faster than the other type pivot selections. On the other hand, if the array is already ordered (increasing or decreasing) selecting the left or right of the array as a pivot will be resulted in O(n^2) time complexity. However, there is still a lot of runtime difference between the increasing and decreasing part of *Table1* in **quickSort FirstElement** row. The reason for that is the program does a swap operation in each recursive call while working on decreasing the ordered array. There is no such operation in increasing the ordered array. However, selecting the pivot as the First or Last Element is still the worst option for ordered arrays. In theory, the best option for ordered arrays would be a middle indexed pivot in quick sort (it is also better than mergeSort in that case). By examining *Table1* we see that the theory holds.

In summary, **quicksort** is an efficient sorting algorithm with an average time complexity of O(n*logn). However, the worst-case time complexity can be O(n^2) if the pivot selection is not optimal. Therefore, choosing a good pivot element is crucial to the efficiency of the algorithm.

Comparison of quickSort and mergeSort:

Mergesort is a stable sorting algorithm that divides the input array into two halves, sorts each half recursively, and then merges the two sorted halves to produce the final sorted output. Mergesort has a worst-case time complexity of O(n*logn), which makes it suitable for sorting large arrays. However, Mergesort requires additional memory space proportional to the size of the input array, which can be a limiting factor in memory-constrained environments.

Quicksort is an unstable sorting algorithm that selects a pivot element from the input array and partitions the array into two sub-arrays based on the pivot element. The pivot element is placed in its final position in the sorted array, and the two sub-arrays are sorted recursively. Quicksort has a worst-case time complexity of O(n^2), but it has an average-case time complexity of O(n*logn), making it faster than Mergesort for small to medium-sized arrays. Quicksort also uses less additional memory than Mergesort.

In summary, Mergesort is generally more suitable for large arrays that can fit into memory, whereas Quicksort is more efficient for smaller arrays and has lower memory requirements. Both algorithms have their own strengths and weaknesses, and the choice between them depends on the specific requirements of the sorting task at hand.

Problems and solution strategies (3.a and 3.b in the assignment)

(Solution domains are limited with the experiment we study on)

3.a) Each character has a numeric ASCII value. So, we can simply say that a word is a number that has multidigit (E.g. “and” word has 3 digits ‘a’=97, ‘n’=110, ‘d’=100. Numeric value of “and”=97*10^2 + 110*10^1 + 100*10^0). Hence we can say alphabetically sorted means sorted in increasing order. A Turkish-English dictionary is alphabetically sorted (sorted in increasing order) for Turkish words. But we work in the English domain. There is no order in our problem. Thus, we can say our problem is similar to 100000 sized arrays of random numbers.

According to *Table1* the best solution for this problem would be **quicksort** algorithm and the pivot should be **FirstElement** or **MidOfFirstMidLast**.

3.b) The list we obtain is ordered from oldest to youngest in our problem. Old to young order is similar to ascending order according to birth dates. To make it young to old we have to make the list in descending order according to birth dates. In ordered lists, the **quick** sort that accepts the middle index as a pivot is the best option. However, pivoting with the middle index is not our solution domain. Instead, we have **MidOfFirstMidLast** pivot type that selects mid of first,mid,last element. This method always selects the pivot as the middle index while working on ordered arrays.

According to *Table1* the best solution for this problem would be **quicksort** algorithm and the pivot should be **MidOfFirstMidLast**.