Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab Report II

[Code No: COMP 314]

Submitted by:

Prashant Manandhar

Roll No:30

Group: Computer Engineering

Level: III year/II sem

Submitted to:

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

Submission Date:22/05/2024

Task I - Quick Sort and Merge Sort

Merge Sort: Merge sort is a stable sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array. In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted. The time complexity of the merge sort is O(n logn) in the worst-case scenario, average-case scenario, and best-case scenario.

Quick Sort: Quick sort is unstable sorting algorithm that employs a divide-and-conquer approach. It begins by selecting a pivot element from the array and partitioning the other elements into two sub-arrays: those less than the pivot and those greater than the pivot. The pivot is then placed in its correct position, and the algorithm recursively sorts the sub-arrays. The time complexity of the merge sort is O(n logn) in average-case scenario, and best-case scenario but $O(n^2)$ in worst case scenario.

Task II - Some test cases to test sorting Algorithm

We have generated various tests for the sorting algorithm. the array containing only negative elements, only positive elements, in descending order, in ascending order, duplicate elements, single data, and empty are tested in those two algorithms.

Output:

```
[Running] python -u "e:\6th sem\Algorithm\Lab 2\test_sorting.py"
.......
----------------------------------------------------------------------
Ran 7 tests in 0.000s

OK
```

Task III: Generate some random inputs for your program and apply both insertion sort and insertion sort algorithms to sort the generated sequence of data. Record the execution times of both algorithms for inputs of different sizes. Plot an input-size vs execution-time graph.

The program generated a random number using the random.randint which generates the random number from range -10000 and 10000. About 2000 random numbers were generated. The generated random were sorted in ascending order. The sorted arrary is used for the worst case of quick sort and best case for merge sort.

The random generated array of numbers are passed into the function named “create_best_case_quick_sort_array” to generate the array for the best case of quick sort.In this function, the passed array is first sorted in ascending order and then stoping condition is check i.e start>end. It finds the middle index and recursively processes the left and right subarrays. The middle element is placed at the end of the merged array to simulate choosing the median as the pivot, which ensures optimal partitioning at each step. This process is repeated for each subarray.

For the worst case of merge sort, function named “create_worst_case_merge_sort_array” is used. This function starts by recursively splitting the input array into smaller subarrays until each subarray contains only one element. Then, it merges these subarrays back together by alternating elements from the left and right subarrays.

The random arrays for general case of the merge, insertion, selection and quick sort. For the best case of merge sort, random array in ascending order is used and for the best case of the quick sort, calculated array for the best case of the quick sort is used. For the worst case of the merge sort, the calculated array for the worst case of the merge sort is used whereas for the worst case of quick sort, the random array in asecendiing order is used. The execution times of all algorithms were recorded for inputs of different sizes, and an input-size vs execution-time graph was plotted.

Observation:

From the graph mentioned below, we can see that Quick Sort performs better than Merge Sort across the entire range of array sizes in both general and best case. The time taken by Quick Sort grows more slowly compared to Merge Sort. The time complexity of both algorithm in both cases (general and best case) is O(n logn). But in case of worst case, Quick Sort performance degrades significantly where its time complexity becomes O(n^2) whereas the performance of merge sort remains stable and predictable, showing its O(n log n) time complexity.

Algorithm	Worst Case	General Case	Best Case
Quick Sort	$O(n^2)$	O(n logn)	O(n logn)
Merger Sort	O(n logn)	O(n logn)	O(n logn)

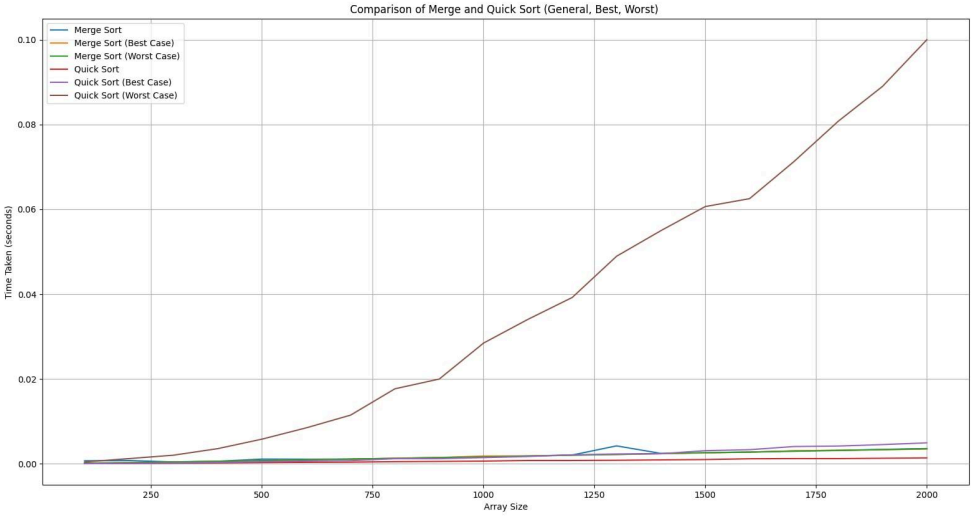
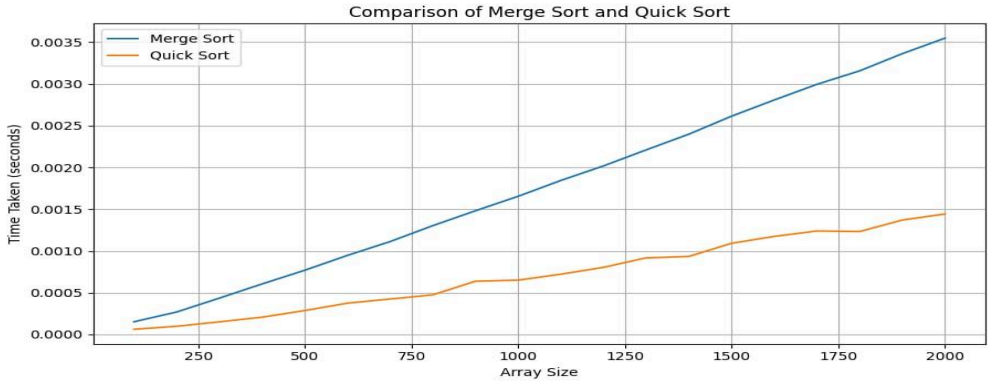
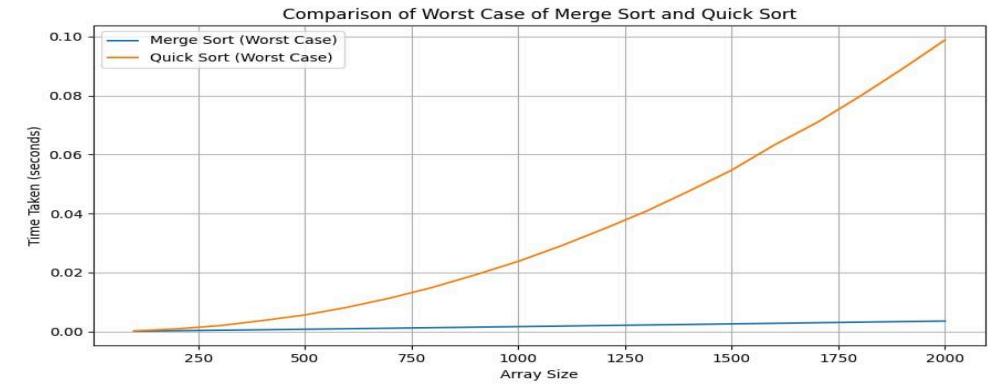
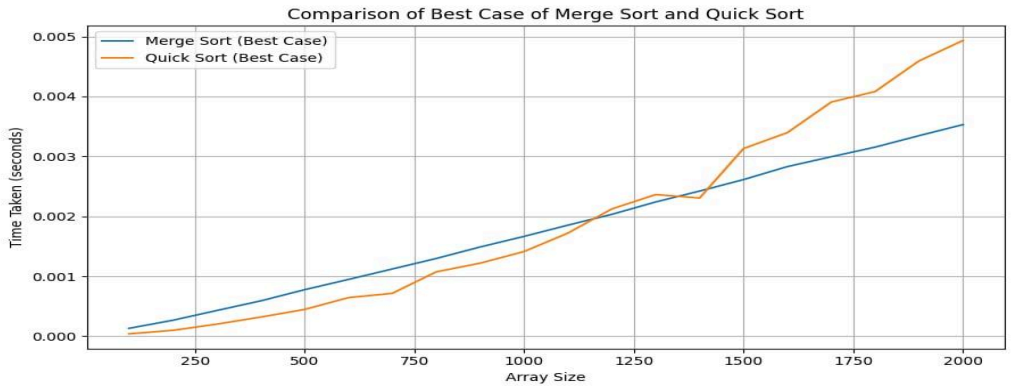


Fig: Graph of Best, worst and general case of Merge and Quick Sort


Comparison of Merge Sort and Quick Sort
Merge Sort
Quick Sort
Time Taken (seconds)
Array Size


Comparison of Worst Case of Merge Sort and Quick Sort
Merge Sort (Worst Case)
Quick Sort (Worst Case)
Time Taken (seconds)
Array Size


Comparison of Best Case of Merge Sort and Quick Sort
Merge Sort (Best Case)
Quick Sort (Best Case)
Time Taken (seconds)
Array Size


Comparison of Insertion Sort and Selection Sort
Insertion Sort
Selection Sort
Time Taken (seconds)
Array Size

Task IV: Compare your results with those from Lab l, and explain your observations.

The graph illustrate the performance comparison between Quick Sort, Merge Sort, Selection Sort, and Insertion Sort under various conditions. The graph shows that the Quick sort is generally fastest in the best case and average case. But its performance drastically decrease in the case for the worst case. For the merge sort, it show stable and predictable performance across all cases with its O(n logn) time complexity. It performs little less efficient when compare to the the Quick sort.

Selection Sort and Insertion Sort both exhibit quadratic time complexity (O(n^2)), making them inefficient for larger datasets. However, between the two, Insertion Sort tends to perform better for smaller arrays due to fewer swaps, especially when the input is nearly sorted. Selection Sort consistently makes n-1 swaps, regardless of the initial order. In summary, Quick Sort is generally the fastest but can be unpredictable, Merge Sort offers consistent performance, and both Selection and Insertion Sorts are less efficient for large arrays but can be useful for small or nearly sorted datasets.

Algorithm	Worst Case	General Case	Best Case
Quick Sort	$O(n^2)$	O(n logn)	O(n logn)
Merger Sort	O(n logn)	O(n logn)	O(n logn)
Insertion Sort	$O(n^2)$	$O(n^2)$	O(n)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

GitHub Link:

The code of the Lab 2 can be found in the following GitHub Link:

https://github.com/Eemayas/Algorithm-and-Complexity-CE2020-Roll-30-COMP-314-Labs/tree/main/Lab2_Manandhar_Prashant_Roll_30_CE