**Data Structures and Algorithms**

Logo, company name

Description automatically generated

**Lab report: 9**

|  |  |
| --- | --- |
| **Name:** | **Ali Salman** |
| **Reg no:** | **FA22-BCE-005** |
| **Class:** | **BCE-3A** |
| **Lab Instructor:** | **Dr. Ali Mustafa** |

**Lab 09**

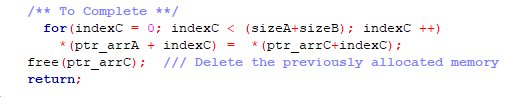
**Quick and Merge Sort Implementation**

**In-Lab Tasks:**

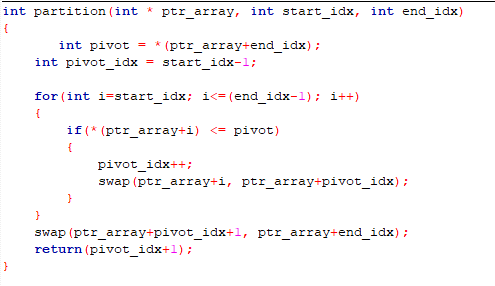
You are given skeleton code for this lab. Your task is to complete the merge () and partition ()

functions for Merge Sort and Quick Sort respectively.

**Programs:**

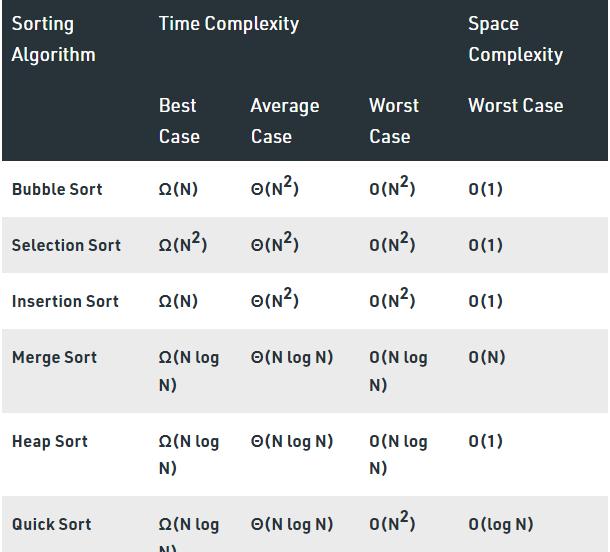


Here is the code for partition function which is used in quick sort algorithm:



**Post Lab:**

Study and perform comparative analysis between different sorting algorithms we have implemented in current and previous Lab.



In comparison-based sorting, elements of an array are compared with each other to find the sorted array.

**Bubble sort and Insertion sort:**

Average and worst-case time complexity: n^2

Best case time complexity: n when array is already sorted.

Worst case: when the array is reverse sorted.

**Selection sort:**

Best, average and worst-case time complexity: n^2

it is independent of distribution of data.

**Merge sort:**

Best, average and worst-case time complexity: nlogn

it is independent of distribution of data.

**Quick sort:**

It is a divide and conquer approach with recurrence relation:

T(n) = T(k) + T(n-k-1) + cn

Worst case: when the array is sorted or reverse sorted, the partition algorithm divides the array in two subarrays with 0 and n-1 elements. Therefore,

T(n) = T (0) + T(n-1) + cn

T(n) = O(n^2)

Best case and Average case: On an average, the partition algorithm divides the array in two subarrays with equal size. Therefore,

T(n) = 2T(n/2) + cn

Solving this we get, T(n) = O(nlogn).

**Critical Analysis:**

In the provided merge function in lab task 1, these two lines are responsible for copying the sorted elements from the temporary array ptr\_arrC back into the original array ptr\_arrA. Let me break down the process:

for(indexC = 0; indexC < (sizeA+sizeB); indexC++)

\*(ptr\_arrA + indexC) = \*(ptr\_arrC + indexC);

Here's a step-by-step explanation:

1. \*Loop Initialization:\* for(indexC = 0; indexC < (sizeA+sizeB); indexC++) initializes a loop where indexC starts from 0 and iterates until it reaches sizeA + sizeB - 1. This loop is responsible for traversing all elements in the temporary array ptr\_arrC.

2. \*Copy Elements:\* \*(ptr\_arrA + indexC) = \*(ptr\_arrC + indexC) copies each element at the current index indexC from the temporary array ptr\_arrC to the corresponding index in the original array ptr\_arrA.

In simpler terms, after merging and sorting the elements in ptr\_arrC, these two lines ensure that the sorted elements are transferred back to the original array ptr\_arrA. The loop goes through each index in the temporary array and copies the sorted element at that index to the corresponding index in the original array, effectively updating the original array with the sorted values.

In this lab, we deeply understand the time complexity of various sorting algorithm and also builds the functions for merge and quick sort algorithm. Both are used for different size and structure of data. The worst-case complexity of quick sort is O(n2) as there is need of lot of comparisons in the worst condition. In merge sort, worst case and average case has same complexities O (n log n). Usage with datasets: Merge sort can work well on any type of data sets irrespective of its size (either large or small).