CCT College Dublin

Algorithms & Constructs

Module Title:	**Algorithms & Constructs**
Assessment Title:	**Integrated / Individual**
Lecturer Name:	**Muhammad Iqbal**
Student Full Name:	**Mohamed jnah**
Student Number:	**2024468**
Date Issued:	**Week of 7th April 2025**
Due Date (Deadline):	**Saturday 10th May at 11.59pm**

Report

Sorting Algorithm :

Merge Sort

To sort our employees was a job for merge sort. Here's why: The number one reason is that merge sort gives us the same performance no matter what -it takes O(n log n) time in all cases, whether best-case, average-case, or worst-case. For our staff management system (where the format of such an input file is not in our control), this reliability is important. Another big advantage of the merge sort is that it is stable. It maintains the same order of employees with the same name more important than you might realize. Suppose you have two "Jassica Lee" entries - with merge sort you know that after sorting their relative order remains the same, so if there was any important order (like date of birth) that information was not lost. I liked that I could easily swap in linked list-based data structures in the future instead of ArrayLists when I realized that merge sort would still operate appropriately with them. The recursive inductive nature of the algorithm also lent itself to such a test - the problem could be recursively divided into manageable descendent problems; their sequential solution and reassembly was a far more elegant construction which could be easier to generate test paths for than an iterative approach.

Searching Algorithm :

Binary Search

To search at all in our employee data it would be natural to use binary search with our sorted employee data: Now, efficiency is the big win here O(n log n) efficiency is much faster than a linear search, and the bigger our employee database gets, the faster it will be. Given that we’re sorting with merge sort, it’s quite natural to be able to use that organization for quick lookups. I also like that binary search is predictable in terms of performance - we get the same relative cost for a query whether our keys are balanced or not (unlike, back in the day, with hash lookups for very similar reasons). The recursive approach also really jived with my project, though I am sure the code wouldn't have been too difficult to wrap my head around if it had been done in an iterative way. Binary search is fine when we’re looking for an exact match, but I also made sure that our implementation present nearby matches when those names happen to be close to one another – not just some rinky-dink stunt, but a real practical feature for a system that is more useful in practice,