

## CCT College Dublin Continuous Assessment

<b>Programme Title:</b>	<i>H.Dip. in Computing</i>		
<b>Delivery Mode:</b>	FT		
<b>Cohort Details:</b>	Feb 2025 cohort		
<b>Module Title(s):</b>	Algorithms & Constructs Software Development Fundamentals		
<b>Assignment Type:</b>	<i>Integrated / Individual</i>	<b>Weighting(s):</b>	Algorithms & Constructs – 60% Software Development Fundamentals – 55%
<b>Assignment Title:</b>	<i>System Modelling &amp; Build</i>		
<b>Lecturer(s):</b>	<i>Ken Healy / Taufique Ahmed</i>		
<b>Issue Date:</b>	<i>28<sup>th</sup> Oct 2025</i>		
<b>Submission Deadline Date:</b>	<i>Saturday 29<sup>th</sup> Nov at 11.59pm</i>		
<b>Late Submission Penalty:</b>	Late submissions will be accepted up to <b>5</b> calendar days after the deadline. All late submissions are subject to a penalty of <b>10%</b> of the mark awarded. Submissions received more than 5 calendar days after the deadline above <b><u>will not</u></b> be accepted and a mark of 0% will be awarded.		
<b>Method of Submission:</b>	<b>This assignment is submitted via Moodle.</b>		
<b>Instructions for Submission:</b>	<p><b><u>Algorithms &amp; Constructs</u></b></p> <p>A GitHub repo must be created. The Netbeans project and report Word document must be put into a GitHub repo for version control. The lecturer must be added to the repository as a collaborator. The GitHub repo link will be added at the end of the report. There should be at least 10 to 15 commits throughout the time worked on the project. Submit your Netbeans .java files individually, and make sure all files contain the same package name, CA_2.</p> <p>If your submission does not load or run, then you may receive a zero mark! Include a section (max 500 words) in your Report submission that clearly explains your choices for the sorting &amp; searching algorithms that you have used.</p> <p><b><u>Software Development Fundamentals</u></b></p>		
	<p>Your design diagrams, user stories, etc. must be submitted in one document in <b>PDF format</b> to the SDF Moodle page. You do not need to include this in your GitHub Repo.</p> <p>There will be an IN-CLASS Q&amp;A held on Tues 2<sup>nd</sup> Dec. You may be asked to explain any part of your submission. Late submissions will be required to attend a separate Q&amp;A in week commencing 8<sup>th</sup> Dec.</p>		
<b>Feedback Method:</b>	<b>Results posted in Moodle gradebook</b>		
<b>Feedback Date:</b>	<i>Following release of approved results by the college</i>		

## **Introduction**

The project deals with designing a system known as the Department Store Employee Management System, using the Java programming language in the NetBeans development environment. The system shall be capable of storing, organizing, and retrieving employee information like the employee's name, type of manager, department, and every piece of relevant information in an effective and reliable way.

In large and complex systems, such as department store chains, efficient management of employee information is the key to rapid decision-making, manpower planning, and administrative report generation. To this end, sorting and searching algorithms are used to ensure that performance will be consistent despite the growth of databases.

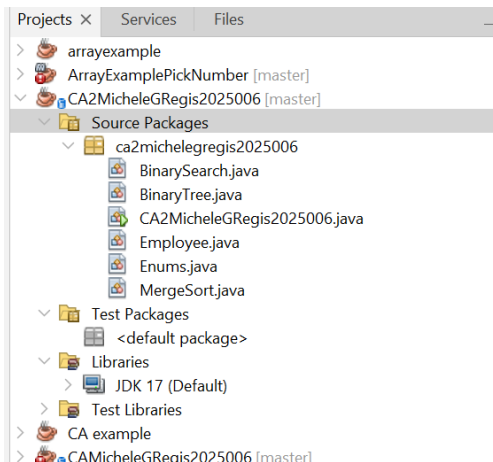
Version control is also used on this project through GitHub, which provides tracking of changes, facilitates collaboration, and offers a record of the history of development. This report describes the chosen algorithms- Merge Sort for sorting and Binary Search for searching-and gives justifications for each selection based on performance, scalability, and reliability. The system was designed to be expandable, allowing for future implementations such as secondary sorting, range queries, and graphical or reporting interfaces.

## **GitHub-**

<https://github.com/MicheleRegis/CA2MicheleGRegis2025006.git>

# Algorithms and Constructs

## The Department Employee Management System



## Sorting Algorithm: Merge Sort

### Main Reason

I chose Merge Sort for the recursive sorting algorithm for this department store system because it guarantees an  $O(n \log n)$  time complexity in all cases: best, average, and worst. Unlike Quick Sort, it doesn't degrade to  $O(n^2)$  with poorly chosen pivots or already-sorted data. Merge Sort exhibits predictable, reliable performance irrespective of input patterns, which is very important in a production environment serving department store managers.

```
run | debug | profile | team | tools | window | help | CA2MicheleGRegis2025006 - Apache NetBeans... | Search (Ctrl+F)
[default config]
CA2MicheleGRegis2025006.java X MergeSort.java X
Source History
49  /**
50  * but it's NOT stable and harder to implement recursively cleanly.
51  */
52  public class MergeSort {
53
54      /**
55       * Public method to start recursive merge sort
56       * @param employees Array of employees to sort
57       * @param left Starting index
58       * @param right Ending index
59       */
60      public static void sort(Employee[] employees, int left, int right) {
61          // Base case: array of size 1 or 0 is already sorted
62          if (left < right) {
63              // Find the middle point to divide the array into two halves
64              int middle = left + (right - left) / 2;
65
66              // Recursively sort the first half
67              sort(employees, left, middle);
68
69              // Recursively sort the second half
70              sort(employees, middle + 1, right);
71
72              // Merge the sorted halves
73              merge(employees, left, middle, right);
74          }
75      }
76  }
```

```
Debug | Profile | Team | Tools | Window | Help | CA2MicheleGRegis2025006 - Apache NetBeans... | Search (Ctrl+F)
[default config]
MergeSort.java X BinarySearch.java X BinaryTree.java X Employee.java X Enums.java X
Source History
10  /**
11  *
12  */
13  public class Enums {
14      /**
15       * Enum for Menu Options
16       * Provides type-safe menu navigation
17       */
18      enum MenuOption {
19          SORT("Sort Employees"),
20          SEARCH("Search Employee"),
21          ADD_RECORDS("Add New Employee"),
22          CREATE_TREE("Create Employee Hierarchy"),
23          EXIT("Exit System");
24
25          private final String displayName;
26
27          MenuOption(String displayName) {
28              this.displayName = displayName;
29          }
30
31          public String getDisplayName() {
32              return displayName;
33          }
34      }
35
36      /**
37       * Enum for Manager Types in Department Store
38       * Represents different management levels
39       */
40  }
```

## Key Advantages

Another important aspect is stability. The Merge Sort algorithm is a stable sort, meaning that equal elements remain in their original order. This becomes useful later if we extend our system to include secondary sorting criteria, such as employee ID or hire date.

The problem fits perfectly with the divide-and-conquer recursive approach. Merge Sort naturally splits the problem into smaller subproblems, sorts them recursively, and then merges the results. This is a very elegant recursive structure that makes the code maintainable, with proper algorithm design principles in mind.

## Scalability Considerations

With the increase of department stores, employee databases grow from hundreds to possibly thousands of records. This  $O(n \log n)$  complexity of Merge Sort makes the system scale well. If the number of employees is doubled, the time taken increases a bit more than double-not that quadratic explosion as in the simpler  $O(n^2)$  algorithms like Bubble or Insertion Sort.

## Trade-Offs Acknowledged

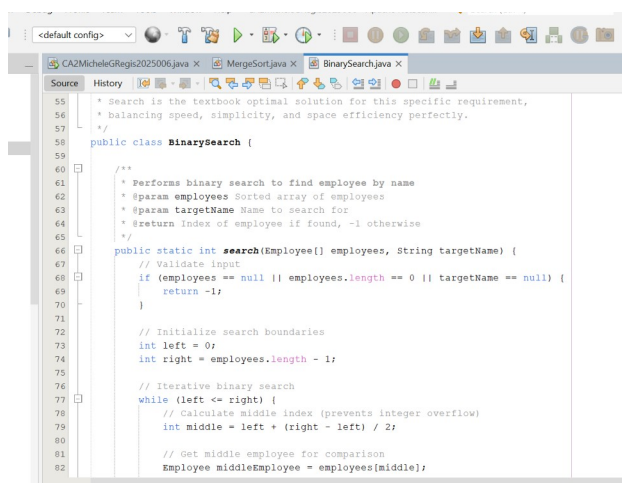
The main trade-off here is space complexity: Merge Sort requires  $O(n)$  auxiliary space for the merging operation. However, on modern systems, this is usually acceptable given the reliability benefits that come with it. I have taken the approach of sacrificing memory to keep up consistent performance. Besides, the extra memory consumption is temporary and freed right after the sorting is done.

## Searching Algorithm: Binary Search

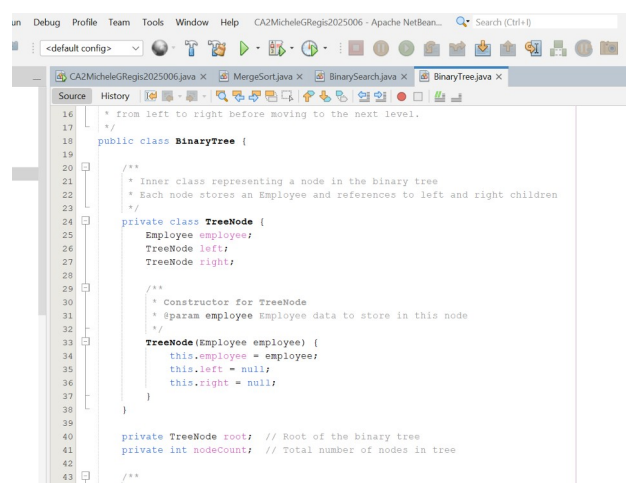
### Core Reasoning

Binary search was the obvious choice for looking up employees, since it provides a time complexity of  $O(\log n)$ .

Since our data is already sorted through Merge Sort, Binary Search takes advantage of this structure to locate employees with logarithmic efficiency. For 1,000 employees, it only takes roughly 10 comparisons versus the average of 500 comparisons in Linear Search.



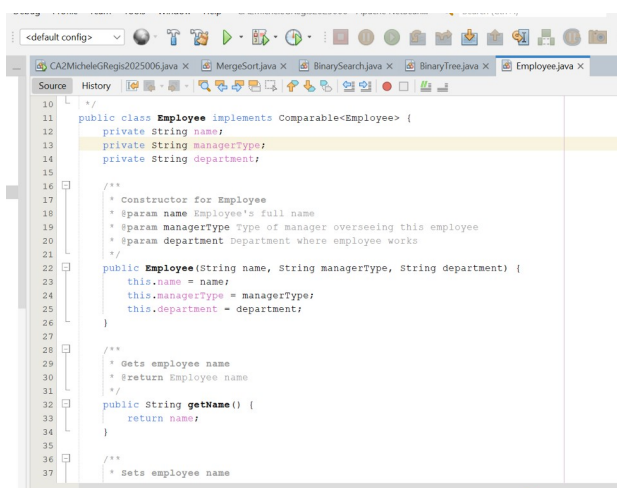
```
55  * Search is the textbook optimal solution for this specific requirement,
56  * balancing speed, simplicity, and space efficiency perfectly.
57  */
58  public class BinarySearch {
59
60      /**
61       * Performs binary search to find employee by name
62       * @param employees Sorted array of employees
63       * @param targetName Name to search for
64       * @return index of employee if found, -1 otherwise
65       */
66      public static int search(Employee[] employees, String targetName) {
67          // Validate input
68          if (employees == null || employees.length == 0 || targetName == null) {
69              return -1;
70          }
71
72          // Initialize search boundaries
73          int left = 0;
74          int right = employees.length - 1;
75
76          // Iterative binary search
77          while (left <= right) {
78              // Calculate middle index (prevents integer overflow)
79              int middle = left + (right - left) / 2;
80
81              // Get middle employee for comparison
82              Employee middleEmployee = employees[middle];
```



```
16  * from left to right before moving to the next level.
17  */
18  public class BinaryTree {
19
20      /**
21       * Inner class representing a node in the binary tree
22       * Each node stores an Employee and references to left and right children
23       */
24      private class TreeNode {
25          Employee employee;
26          TreeNode left;
27          TreeNode right;
28
29          /**
30           * Constructor for TreeNode
31           * @param employee Employee data to store in this node
32           */
33          TreeNode(Employee employee) {
34              this.employee = employee;
35              this.left = null;
36              this.right = null;
37          }
38
39          private TreeNode root; // Root of the binary tree
40          private int nodeCount; // Total number of nodes in tree
41
42          /**
```

### Practical Impact

Speed counts in a busy department store where managers frequently query employee information for scheduling, performance reviews, or department assignments. Similarly, Binary Search provides instant results even as the number of staff grows, which again improves user experience and operational efficiency.

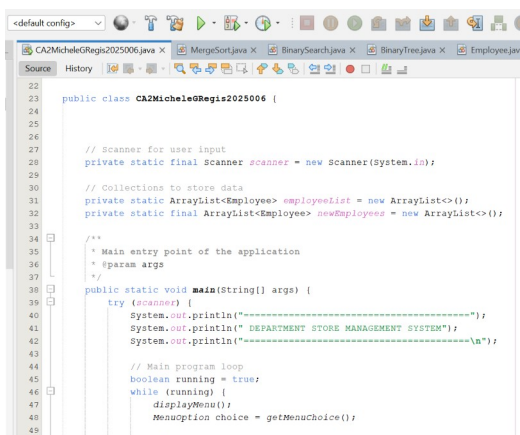


```
10  /**
11  */
12  public class Employee implements Comparable<Employee> {
13      private String name;
14      private String managerType;
15      private String department;
16
17      /**
18       * Constructor for Employee
19       * @param name Employee's full name
20       * @param managerType Type of manager overseeing this employee
21       * @param department Department where employee works
22       */
23      public Employee(String name, String managerType, String department) {
24          this.name = name;
25          this.managerType = managerType;
26          this.department = department;
27      }
28
29      /**
30       * Gets employee name
31       * @return Employee name
32       */
33      public String getName() {
34          return name;
35      }
36
37      /**
38       * Sets employee name
```

## Why Not Alternatives?

Linear Search is rejected, though simple, because it wastes the sorted structure that we maintained with such care. For very large datasets, this would be noticeably slow.

Hash tables might seem faster, averaging  $O(1)$ ; but they require  $O(n)$  extra space for the hash structure and don't support features we may want in future versions-such as finding "closest matches" for partial name searches or doing range queries for alphabetical sections.

A screenshot of an IDE window showing a Java file named CA2MicheleGRegis2025006.java. The code defines a public class with a main method. It uses Scanner for input, ArrayList for data storage, and a while loop for a menu-driven program. The code is as follows:

```
22 public class CA2MicheleGRegis2025006 {
23
24
25
26
27     // Scanner for user input
28     private static final Scanner scanner = new Scanner(System.in);
29
30     // Collections to store data
31     private static ArrayList<Employee> employeeList = new ArrayList<>();
32     private static final ArrayList<Employee> newEmployees = new ArrayList<>();
33
34     /**
35      * Main entry point of the application
36      * @param args
37      */
38     public static void main(String[] args) {
39         try (scanner) {
40             System.out.println("=====");
41             System.out.println(" DEPARTMENT STORE MANAGEMENT SYSTEM");
42             System.out.println("=====\\n");
43
44             // Main program loop
45             boolean running = true;
46             while (running) {
47                 displayMenu();
48                 MenuOption choice = getMenuChoice();
49
50             }
51         }
52     }
53 }
```

## Algorithm Synergy

Binary Search and Merge Sort go together perfectly. Merge Sort creates the required sorted structure that Binary Search needs. This design shows an understanding of algorithm relationships and dependencies in data structures.

## Conclusion

Both algorithms were chosen for reliability, scalability, and performance guarantees rather than clever optimizations that might fail under specific conditions. This conservative, production-ready approach ensures that the system serves department store staff effectively as the organization grows.

GitHub-

<https://github.com/MicheleRegis/CA2MicheleGRegis2025006.git>

## Reference

Sehgal, K., 2018. *A Simplified Explanation of Merge Sort*. Medium, 25 January. Available at:

<https://medium.com/karuna-sehgal/a-simplified-explanation-of-merge-sort-77089fe03bb2> [Accessed 16 November 2025].

Codecademy Team, 2025. *Time Complexity of Merge Sort: A Detailed Analysis*. Codecademy. Available at:

<https://www.codecademy.com/article/time-complexity-of-merge-sort> [Accessed 16 November 2025].

EnjoyAlgorithms, 2020. *Merge Sort – Algorithm, Source Code, Time Complexity*. HappyCoders.eu. Available at:

<https://www.happycoders.eu/algorithms/merge-sort/> [Accessed 16 November 2025].

Crio.Do, 2024. *Everything You Need To Know About Merge Sort*. Crio Blog, 9 September. Available at:

<https://www.crio.do/blog/merge-sort-algorithm-dsa> [Accessed 16 November 2025].

Crio.Do, 2022. *Understanding Binary Search Algorithm*. Crio Blog. Available at:

<https://www.crio.do/blog/understanding-binary-search-algorithm/> [Accessed 16 November 2025].

EnjoyAlgorithms, 2023. *Binary Search Algorithm – EnjoyAlgorithms*. EnjoyAlgorithms Blog. Available at:

<https://www.enjoyalgorithms.com/blog/binary-search-algorithm/> [Accessed 16 November 2025].

InterviewKickstart, 2025. *Binary Search Algorithm: How It Works and Why It's Efficient*. InterviewKickstart Blog.

Available at: <https://interviewkickstart.com/blogs/learn/binary-search> [Accessed 16 November 2025].