# HospitalApp Project for Algorithms:

### Introduction

The HospitalApp was developed as a command-line interface (CLI) application strictly following the instructions: no GUI components like Java Swing or JavaFX and no use of external build tools like Maven or Gradle. The goal was to create a robust, well-structured program that demonstrates strong programming practices and object-oriented principles.

### Use of Enums

The program makes strong use of **enums**. For example, in MenuOption.java, it defines the menu choices in an organized way; in ManagerType.java, it lists the different types of managers; and in Department.java, it holds the list of hospital departments.  
Using enums helps make sure that only valid options are used in the program, avoids mistakes or random entries, and makes the code easier to read and maintain. It also fits well with the project’s goal of having clear, structured menus and predefined choices for the user.

### Object-Oriented Inheritance

The design uses **inheritance** by creating a main class, Person.java, which holds the basic information like the name. Then, Employee.java builds on top of it by adding extra details such as the manager type and department.  
This setup avoids repeating the same code in multiple places, makes it easier to expand the system in the future, and follows the object-oriented programming principles that were recommended in the assignment.

## ArrayList as Data Storage

An ArrayList is used in HospitalApp.java to store dynamic lists of employees. This choice was made because **ArrayLists** allow efficient index-based access, dynamic resizing, and seamless integration with sorting and searching algorithms without needing external dependencies. Using Java’s core collection classes respects the project's restrictions against external libraries.

### Sorting Algorithm Choice

The project uses a **recursive insertion sort** to sort the data loaded from Applicants\_Form.txt, implemented in SortUtil.java. Recursive insertion sort was selected primarily because it meets the project’s requirement for a recursive sorting algorithm while offering **simplicity and clarity of implementation** (Weiss, 2012) Compared to more complex recursive sorts such as **merge sort** or **quick sort**, insertion sort has lower overhead and is easier to debug and maintain, especially for small datasets. While merge sort provides **O(n log n)** performance (Cormen, 2009), its use of additional memory for temporary arrays was unnecessary given the project’s dataset size. In contrast, insertion sort operates in-place with **O(1) space complexity**, making it more memory-efficient for a small number of records (Robert Sedgewick , Kevin Wayne, 2011). Although insertion sort has a worst-case time complexity of **O(n²)**, its performance is acceptable for this context because the list contains fewer than 100 records, where the differences between O(n²) and O(n log n) are negligible (Donald E. Knuth, 2011).

### Searching Algorithm Choice

For **searching,** the project employs a **sequential (linear) search**, also implemented in SearchUtil.java. Sequential search was chosen for its **simplicity and flexibility**, as it works on both **sorted and unsorted lists** and does not require any preprocessing (Mark Allen Weiss, 2007). Given that the dataset is relatively small and changes dynamically (e.g., new applicants can be added), a sequential search avoids the need to maintain a sorted list, which would be required for a binary search. Although **binary search** offers **O(log n)** time complexity on sorted data (Cormen, 2009), maintaining sorted order would introduce extra complexity and cost in this scenario, as every insertion or deletion would require re-sorting or additional algorithms like insertion in a sorted list. Furthermore, for small lists (e.g., under 100 elements), the difference in practical search time between **O(n)** and **O(log n)** is minimal and not user-perceptible (Robert Sedgewick , Kevin Wayne, 2011). Sequential search also provides **greater flexibility** in handling future requirements, such as searching by multiple attributes, without enforcing strict ordering constraints.

In conclusion, the use of **recursive insertion sort** and **sequential search** aligns with the project’s educational goals, dataset size, and simplicity requirements. These algorithms provide sufficient performance while minimizing implementation complexity and resource usage. More advanced algorithms like merge sort or binary search would only add **unnecessary complexity and memory overhead** without significant performance gains for the dataset’s scale.

### User Input and Validation

Adding new users is interactive: the user types in a name, picks a manager type, and selects a department. The Validation.java class makes sure the name only has valid characters (letters and spaces), helping to stop bad or incorrect data from getting into the system. This step supports the project’s goal of keeping the data clean and reliable.

### Random Data Generation

For testing, the system can generate random employees using DataGenerator.java, which combines predefined names with random manager types and departments. This ensures the generated data is valid and complies with the project’s constraints of only using predefined categories.

### Saving Data to File

The program allows saving the current list of employees to a file like list.txt, ensuring persistence of data across sessions, which is critical for real-world applications and adheres to good programming practice.

### **GitHub Repository Link**: <https://github.com/Luzinha2024/HospitalApp-CA2-LucianeHoshino.git>

### **Video Link on YouTube**: <https://youtu.be/o0G17k9nYwc>

### References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.  
Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.  
Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.  
Weiss, M. A. (2014). *Data Structures and Algorithm Analysis in Java* (3rd ed.). Pearson.