Explain the Big O complexity of your priority and sorting mechanisms.

First, we implemented a prioritization system in the hospital. We used both normalQueue and priorityQueue for this system. Therefore, when a patient first arrived, we checked whether it was an emergency or not. If it is an emergency patient, it is placed in the emergency queue and we do not scan the general list when using this system, Enqueue so the time complexity is **O(1)**.

Similarly, when starting treatment, we look at the patient's place in the queue, so in this process it is **O(1)**.

Sorting Mechanism - Bubble Sort

When using a ranking algorithm at the hospital, we also sorted patients according to the severity of their illness. This sorting algorithm uses nested loops. Each element in the list is being compared. Therefore, the time complexity is **O(n^2)**.

How would using a heap-based priority queue improve performance?

My code works for 2 levels. Normal and Prioritiy. That's why using a queue makes more sense. Queue works fast. If the hospital had more priority levels, using the queue would be a bit problematic. I need to go through the entire list item by item. That will naturally take a lot of time. If I used **heap**, I wouldn't need to search to find an emergency patient. Because the heap kept the largest number, meaning the emergency room in the hospital, at the top. Therefore, admitting and treating patients would be much faster and easier.

Detailed Design, Algorithms, and Performance Analysis

In this project, for the CENG201 course, I designed a Hospital Patient Management System using the Java programming language. The main goal of the project was to practice data structures that I learned during the semester. In the system, I used linked lists, queues, stacks and hash tables together.

I handled patient admission operations in the **PatientList** class using a singly linked list. I searched patients by their ID numbers. Because of the linked list structure, the search operation has **O(n)** time complexity.

For managing treatment requests, I used the **TreatmentQueue** class. The queue works with the FIFO logic. To give priority to emergency patients, I used two different queues: one for normal patients and one for priority patients. In this way, priority patients are treated first. Enqueue and dequeue operations work in **O(1)** time.

I managed discharged patients using the **DischargeStack** class. This structure works with the LIFO logic and allows quick access to the most recently discharged patient. Push, pop and peek operations in the stack have **O(1)** time complexity.

To access patient information faster, I used **HashMap**<Integer, Patient> in the system. With this structure, patient search operations can be done in average **O(1**) time.