Temporal Complexity analisis:

Our first analysis will be for the printListByPriorityMethod(), followed by the next code:

|  |  |  |
| --- | --- | --- |
| # | Line | Times executed |
| 1 | System.out.println("===== Tasks ordered by priority ====="); | 1 |
| 2 | if (priorityTask.isEmpty()) { | 1 |
| 3 | System.out.println("No tasks to show."); | 1 |
| 4 | Return;  } | 1 |
| 5 | PriorityQueue<Task> tempQueue = new PriorityQueue<>(); | 1 |
| 6 | while (!priorityTask.isEmpty()) { | n + 1 (Number of stored tasks). |
| 7 | Task task = priorityTask.dequeue(); | n |
| 8 | System.out.println(task.toString()); | n |
| 9 | tempQueue.enqueue(task);  } | n |
| 10 | priorityTask = tempQueue; | 1 |
| 11 | System.out.println("======================================"); | 1 |

Now we must do the summatory of each “Times executed” so we can define the Big O notation:

*T(n)= 1+1+1+1+1+1+1+n+n+(n+1)*

*T(n) = 8 + 4n*

And so, we can say that the algorithm time complexity is: Θ(n)

Second Algorithm:

Our second analysis will be for the removeElement(T element), followed by the next code:

|  |  |  |
| --- | --- | --- |
| # | Line | Times Executed |
| 1 | boolean valid; | 1 |
| 2 | if (element != null) | 1 |
| 3 | Queue<T> temp = new Queue<>(); | 1 |
| 4 | while (!isEmpty()) { | n + 1 |
| 5 | T currentItem = dequeue(); | n \*n -1 (Time complexity of dequeue is n) |
| 6 | if (!currentItem.equals(element)) { | n - 1 |
| 7 | temp.enqueue(currentItem); | n\*n - 1 |
| 8 | while (!temp.isEmpty()) { | n + 1 |
| 9 | enqueue(temp.dequeue()); | n \* n - 1 |
| 10 | valid = !isEmpty(); | 1 |
| 11 | } else {  valid = false;  } | 1 |
| 12 | return valid; | 1 |

Now we must do the summatory of each “Times executed” so we can define the Big O notation:

*T(n) = 1 + 1 + 1 + (n+1) + (n\*n) + n + (n\*n) + (n+1)+1+1*

*T(n ) = 3n2 + 3n + 3*

And so, we can say that the algorithm time complexity is: Θ(n2)