**Reflexión:**

Para esta actividad utilizamos una fila de prioridad que se basa físicamente en un vector para poder acceder a los elementos mediante índices. Esto tiene la ventaja de que podemos extraer, en este caso, el elemento más grande del conjunto en tiempo O(1) ya que se encuentra al principio del vector; sin embargo, como al insertar o borrar un elemento se tiene que reacomodar el vector, la complejidad de estas operaciones es O(log n), que sigue siendo muy rápido, sin embargo, no tanto como hacerlo en una lista enlazada, a pesar de esto, para nuestro objetivo es un sacrificio que vale la pena hacer.

La complejidad final del programa es de O(nlog n) para leer el txt de los datos, O(nlog n) para ordenar los registros por ip, O(n) para exportar estos registros a un txt, O(nlog n) para contabilizar las repeticiones por ip, O(nlog n) para volver a ordenar todas las ip por número de repeticiones, O(n) para exportar estas ips a un txt y por último O(1) para obtener los 5 registros más grandes de esta lista, ya que ya tenemos armado el vector; por lo que tenemos una complejidad final de O(4\*nlogn + 2\*n + 1) = O(nlogn), que se mantiene igual si no tomamos en cuenta la creación de los txt.

Gran parte de por qué es tan buena la eficiencia de este programa es por el uso de la lista de prioridad, que se comporta en parte como un BST, ya que su búsqueda es O(log n) y la complejidad de la búsqueda del elemento más grande es constante. Si lo comparamos con un vector simple, la búsqueda es O(n) y por si no fuera poco, para encontrar el elemento más grande la complejidad también es O(n), por lo que si quisiéramos hacer HeapSort de esta manera obtendríamos una complejidad de O(n^2), que tardaría mucho tiempo con los 16,808 registros que tenemos en nuestro txt.

