Jaime Eduardo López Castro A00833173

Lo que nuestro programa de la actividad 1.3 hace es básicamente que lee los registros de una bitácora creando un objeto de tipo Registro para cada línea y guardándolo en un vector dentro de una clase Bitácora, donde podemos modificarlo después.

Si queremos buscar un elemento dentro de esta bitácora, es más eficiente ordenarla de acuerdo a cierto atributo y luego utilizar una búsqueda binaria para encontrarlo más rápidamente.

En nuestro caso utilizamos MergeSort para ordenar los elementos en el vector con respecto a su Unix Time, en general no es definitivamente mejor que QuickSort, pero su tiempo de ejecución es ligeramente más rápido, es por eso que utilizamos este algoritmo.

Tabla: Complejidad de los algoritmos de ordenamiento.

Algoritmo	Mejor	Promedio	Peor	Estable	Espacio
Swap sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	$O(1)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Sí	$O(n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Sí	$O(\log n)$

Una vez ya tenemos ordenado nuestro vector, es cuestión de utilizar Búsqueda Binaria para dar recorrer el vector en O(log n); como implementamos sobrecarga de operadores en la clase Registro, no hay ningún inconveniente con usar el código que hicimos en actividades anteriores.

Tabla: Complejidad de los algoritmos de búsqueda.

Algoritmo	Mejor	Promedio	Peor
Búsqueda secuencial	$O(1)$	$O(n)$	$O(n)$
Búsqueda secuencial en arreglos ordenados	$O(1)$	$O(n)$	$O(n/2)$
Búsqueda binaria	$O(1)$	$O(\log_2 n)$	$O(\log n)$

Así es como al final, después de haberlo ordenado y luego buscado por búsqueda binaria, obtenemos una complejidad de O(n log n + log n) = O(n log n), mucho mejor que si la hubiéramos buscado elemento por elemento, con complejidad de O(n).