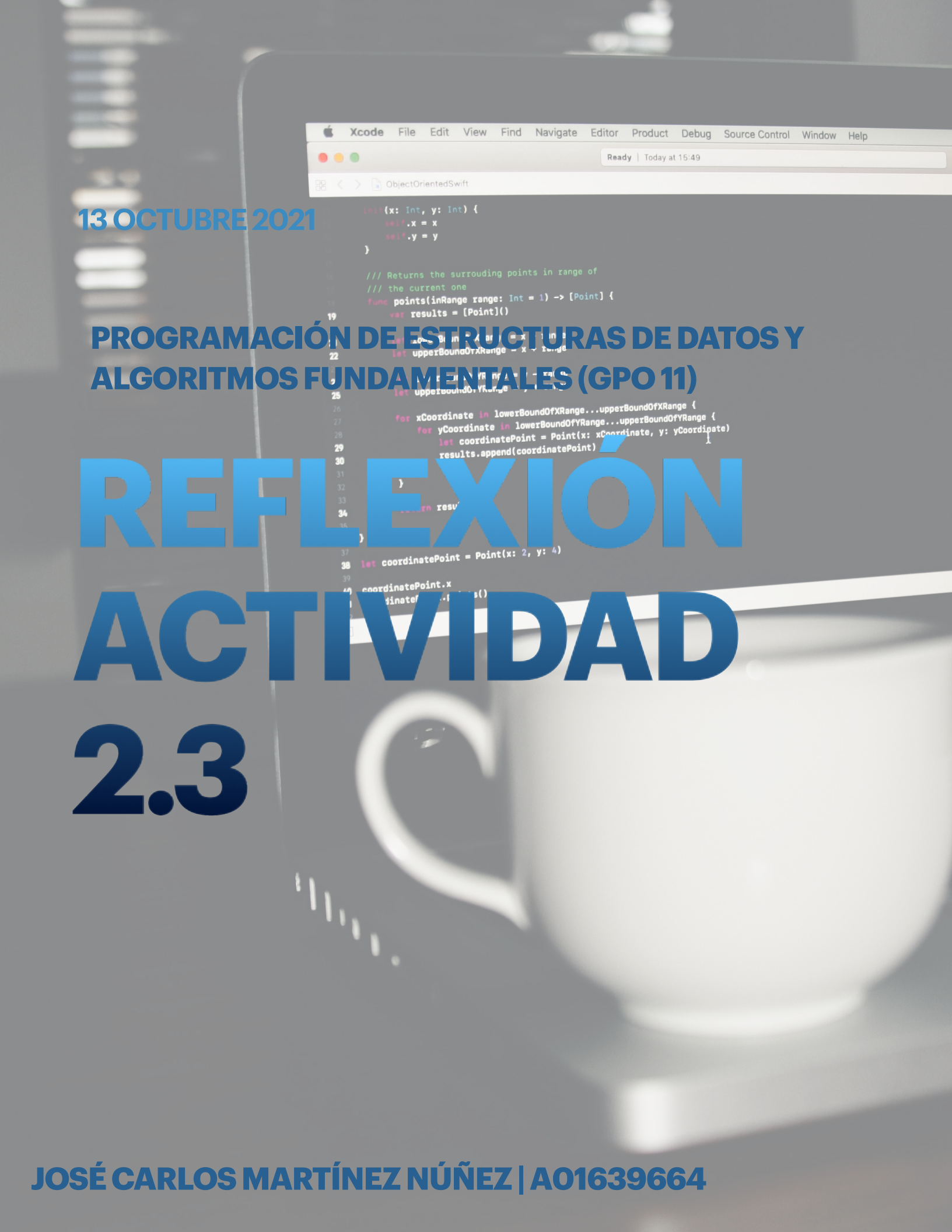
13 OCTUBRE 2021

PROGRAMACIÓN DE ESTRUCTURAS DE DATOS Y ALGORITMOS FUNDAMENTALES (GPO 11)

REFLEXIÓN ACTIVIDAD 2.3



JOSÉ CARLOS MARTÍNEZ NÚÑEZ | A01639664

Índice

PLANTEAMIENTO DEL PROBLEMA

El programa a realizar representa una bitácora con varias entradas. Cada entrada representa una falla en cierto sistema, nuestro programa debe organizar dichas entradas para que sea posible realizar búsquedas dentro de la bitácora y exportar los resultados a un nuevo archivo de texto.

Se nos dio un archivo de texto inicial con varias entradas de la bitácora, este archivo tiene el siguiente formato:

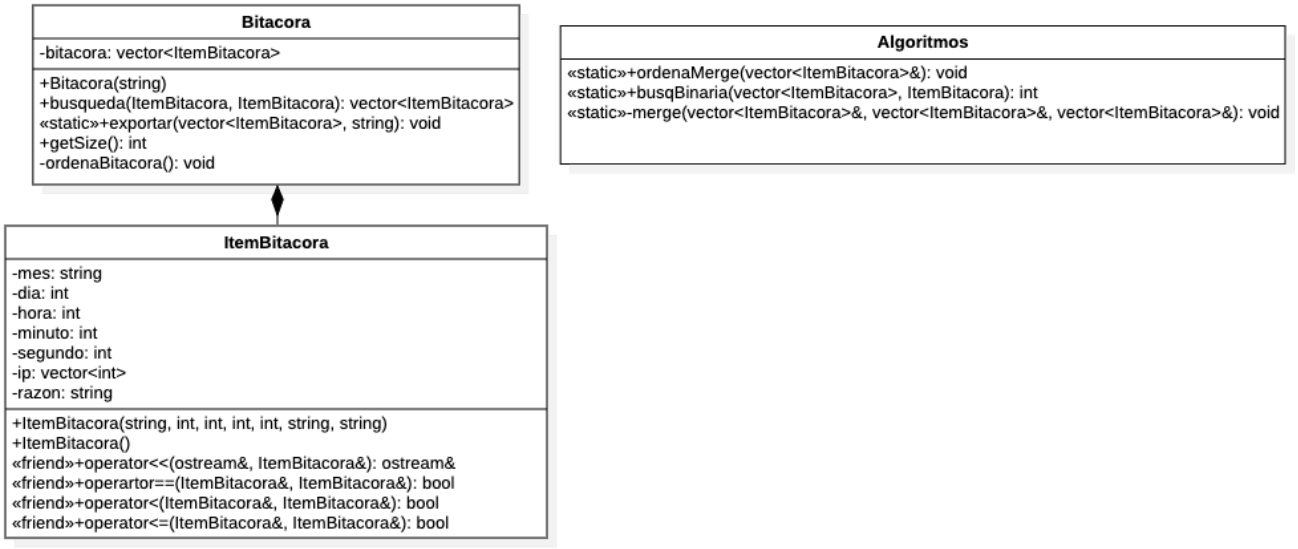
- **Mes** en formato de 3 letras.
- **Día**
- **Hora** en formato hh:mm:ss
- **Dirección IP** origen en formato ###.##.###.###:####
- **Razón** de la falla.

Ejemplo:

```
Aug 4 03:18:56 960.96.3.29:5268 Failed password for admin
```

Las búsquedas se realizarán por medio de la dirección IP, se recibirá un rango de inicio a fin y se deberá imprimir y exportar los resultados en orden descendente.

ESTRUCTURA DE CLASES



Se tomaron en cuenta 3 clases para la realización del programa:

`ItemBitacora`: representa una entrada individual de la bitácora.

`Bitacora`: representa un conjunto de elementos `ItemBitacora`.

`Algoritmos`: cuenta con los métodos de ordenamiento y búsqueda que se utilizan a lo lago del programa.

MÉTODOS DE ORDENAMIENTO Y BÚSQUEDA

Como se menciono en la sección de “Planteamiento de Problema”, en el programa se debe ordenar la bitácora y realizar búsquedas dentro de esta por lo cual debemos escoger un algoritmo para buscar y otro para ordenar los elementos. Algo muy importante a considerar para tomar estas desiciones es el número de elementos con el cual se va a trabajar, para propósitos de este trabajo el archivo *“bitacora.txt”* cuenta con **16,807** lineas por lo cual contamos con ese número de entradas.

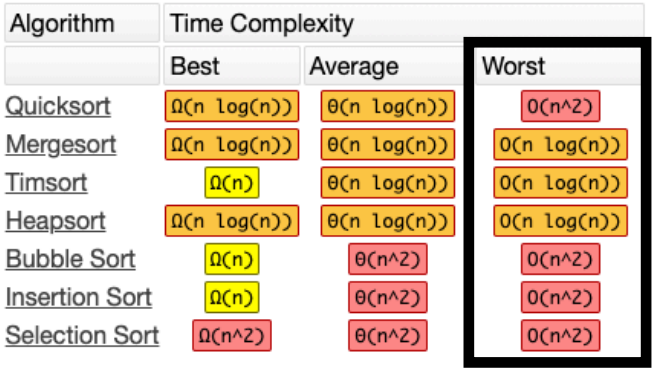
COMPARACIÓN DE MÉTODOS DE ORDENAMIENTO

Para escoger un método de ordenamiento analizamos la complejidad de tiempo de varios algoritmos vistos en clase entre ellos están:

- Ordenamiento por Intercambio
- Ordenamiento Burbuja
- Ordenamiento QuickSort
- Ordenamiento Merge

Considerando la posibilidad de escalar la aplicación para un número de elementos mayor y varios usuarios concurrentes, tomamos en cuenta complejidad en el **peor de los casos** para cada algoritmo.

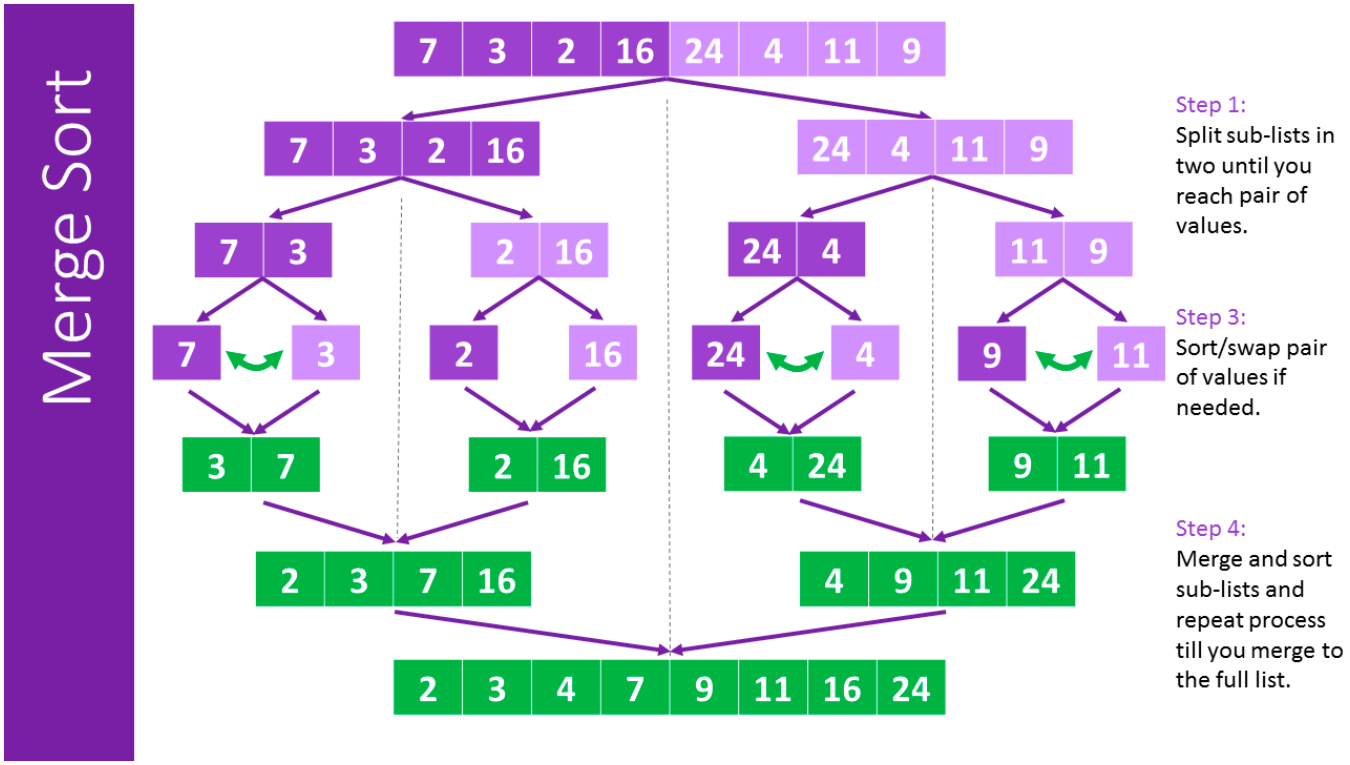
En la tabla adjunta se puede observar como la mayoría de los algoritmos de ordenamiento, en el peor de los casos, tienen una complejidad de tiempo de $O\left(n^2\right)$. Sin embargo podemos ver como el ordenamiento Merge es de los que tienen un mejor desempeño con una complejidad de $O\left(n \log n\right)$.



Algorithm	Time Complexity		
	Best	Average	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Timsort	Ω(n)	θ(n log(n))	O(n log(n))
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)

Big-O Algorithm Complexity Cheat Sheet (Bigocheatsheet.com, 2021)

ORDENAMIENTO MERGE



Merge Sort Algorithm (101 Computing, 2019)

El gran desempeño del ordenamiento Merge nace directamente de su funcionamiento. El método de ordenamiento Merge toma nuestro vector y lo divide a la mitad recursivamente hasta llegar a pares de elementos, después utilizando una función auxiliar ordena los pares, así cada sub-lista queda ordenada. Este proceso como se menciono anteriormente se repite recursivamente hasta que todo el vector se encuentre ordenado.

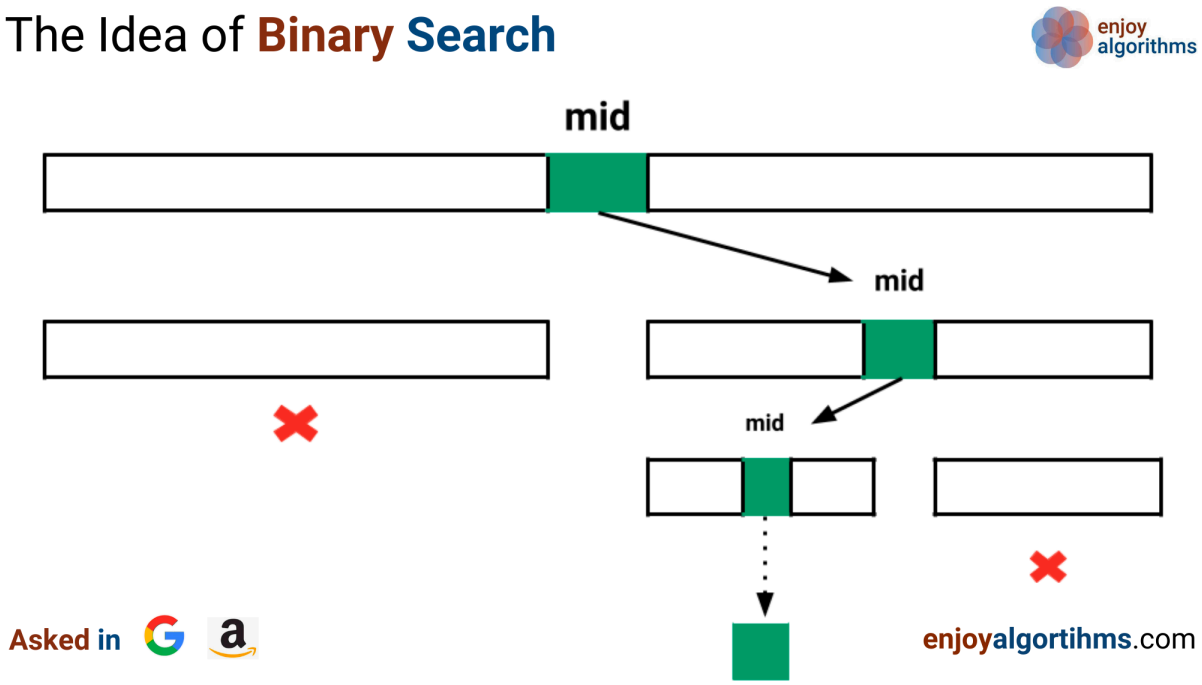
COMPARACIÓN DE MÉTODOS DE BÚSQUEDA

De igual manera para determinar el mejor método de búsqueda hicimos una comparación de los 2 métodos vistos en clase:

- Búsqueda Secuencial/Lineal
- Búsqueda Binaria

Al revisar la complejidad de tiempo de ambos métodos ($O(n)$ y $O(\log n)$ respectivamente) podemos determinar que el método de búsqueda binaria es mucho más efectivo para una mayor cantidad de elementos.

BÚSQUEDA BINARIA



Binary Search Algorithm (Enjoyalgorithms.com, 2020)

Este método parte desde un vector que ya se encuentra ordenado. Se posiciona en medio de este y compara ese valor con el valor a buscar. Si el elemento que estamos buscando es más grande que el elemento de en medio descartamos la primer mitad del vector, sí es más pequeño descartamos la segunda mitad, repetimos ese proceso de eliminación hasta que encontramos el valor deseado.

¿COMO APLICAMOS LOS MÉTODOS?

Tanto para ordenar la bitácora como para realizar búsquedas utilizando los métodos mencionados anteriormente necesitamos una manera en la que podamos comparar dos entradas de bitácora. Para hacer esto se sobrecargaron algunos operadores lógicos (==, <

<=) para que así tomando en cuenta los distintos atributos de cada instancia del objeto podamos hacer comparaciones entre estas.

OPERADOR ==

Esta es la más sencilla de las sobrecargas, sin embargo, es de vital importancia conocer cuando dos instancias de `ItemBitacora` son iguales.

```
bool operator==(const ItemBitacora &x, const ItemBitacora &y)
{
    return (x.mes == y.mes) && (x.dia == y.dia) && (x.hora == y.hora) &&
(x.minuto == y.minuto) && (x.segundo == y.segundo) && (x.ip == y.ip) &&
(x.razon == y.razon);
}
```

Básicamente para determinar esta igualdad entre dos instancias comparamos cada atributo individualmente y sí alguno difiere de una instancia a otra regresamos falso.

OPERADORES < Y <=

Tomando en cuenta que el ordenamiento y las búsquedas serán realizadas por medio de la IP de la entrada en la bitácora ese atributo es el que debemos considerar para realizar estas comparativas. Para facilitar este proceso en lugar de guardar la IP como un string, separamos cada número que la representa y la guardamos en un `vector<int>`.

El proceso que se lleva a cabo para determinar si una IP es menor que otra es el siguiente:

Vector que representa una IP: x				
192	168	0	1	12345

Vector que representa una IP: y				
192	168	1	65	12345

Necesitamos encontrar el primer número donde dos IP's lleguen a diferir, en el presente ejemplo eso sería en `x[2]`y `y[2]`. Una vez que encontrado dónde ambas IP's difieren podemos ignorar el resto de la IP y simplemente regresar el resultado de `x[2] < y[2]`. Por lo cual en este ejemplo podemos decir que la instancia `x` es más pequeña que la instancia `y`. Por último para el operador <= en el caso de que no haya un punto donde ambas IP's difieren este regresa verdadero.

PROCESO DE BÚSQUEDA

El proceso que se lleva a cabo cada vez que se realiza una búsqueda es el siguiente:

1. Se pide la IP de Inicio y la del final del rango.

2. Creamos 2 objetos `ItemBitacora` por cada IP. (El valor del resto de los atributos no tiene importancia)

3. Agregamos los objetos al final vector de `Bitacora`.

Datos del documento "bitacora.txt"	`ItemBitacora` con IP de Inicio	`ItemBitacora` con IP de Fin

4. Ordenamos el vector `Bitacora` utilizando el método de MergeSort.

Datos del documento "bitacora.txt"	`ItemBitacora` con IP de Inicio	Rango deseado por el usuario	`ItemBitacora` con IP de Fin	Datos del documento "bitacora.txt"

5. Utilizando el método de búsqueda binaria obtenemos la posición de los dos objetos que agregamos anteriormente.

6. Creamos un nuevo `vector<ItemBitacora>` correspondiente al rango deseado por el usuario.

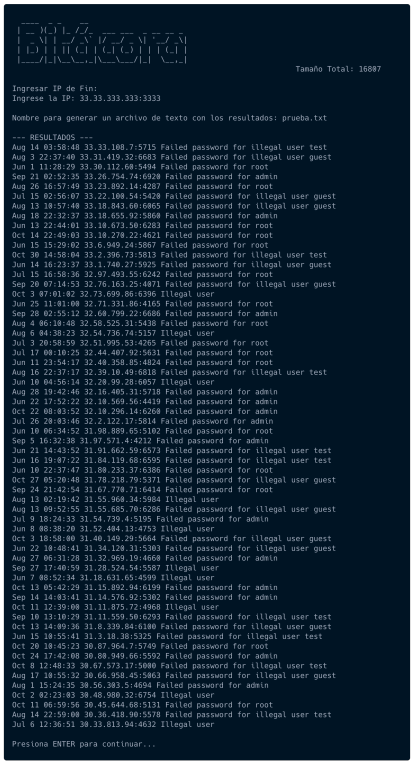
7. Eliminamos del vector `Bitacora` los dos objetos temporales.

8. Imprimimos y guardamos el vector creado en el paso 6.

CASOS DE PRUEBA

Caso de Prueba: IP Inválida

Caso de Prueba: Rango de IP’s Inválido



Caso de Prueba: Rango 30.30.300.300:3000 a 33.33.333.333:3333

REFERENCIAS

Big-O Algorithm Complexity Cheat Sheet. (2021). Bigocheatsheet.com. https://www.bigocheatsheet.com/

Merge Sort Algorithm (2019). 101 Computing. https://www.101computing.net/merge-sort-algorithm/

Binary Search Algorithm: Fast Searching on Sorted Array! (2020). Enjoyalgorithms.com. https://www.enjoyalgorithms.com/blog/binary-search-algorithm