Act 1.3

Importancia y eficiencia del uso de los diferentes algoritmos de ordenamiento y búsqueda

Santiago Amir Rodríguez González
A01739942

1. Introducción

Ordenar datos es una operación esencial en informática: casi toda aplicación que maneja listas, registros o eventos necesita ordenar información en algún momento. La elección del algoritmo de ordenamiento y de las estructuras auxiliares condiciona el rendimiento del sistema y la capacidad de respuesta ante consultas. En el contexto de bitácoras (registros de fechas, horas, direcciones IP y mensajes) este problema se vuelve especialmente relevante: una estructura de datos y algoritmos bien elegidos permiten responder rápidamente a consultas por rango temporal, detectar patrones o agrupar eventos, mientras que una mala elección puede convertir tareas triviales en cuellos de botella.

2. Algoritmos de Ordenamiento

Los algoritmos de ordenamiento son procedimientos diseñados para organizar u ordenar un conjunto de datos o elementos en una forma específica (ascendente o descendente). Este tipo de algoritmos se ocupa mucho en ciencia de datos y programación para mejorar la eficiencia y velocidad de la búsqueda de información.

El estudio de los mismos datos y los varios tipos de ordenamiento nos ayuda a evaluar que tan eficiente será nuestro programa, podemos analizar su complejidad temporal y espacial, y así determinar cuales algoritmos son más rápidos.

2.1 Tipos de algoritmos de ordenamiento

2.1.1 Quick Sort

- Idea: elegir un pivote, particionar el arreglo en elementos menores y mayores, y aplicar recursión.
- Ventaja: alto rendimiento práctico, buena localidad de memoria y baja sobrecarga.
- Complejidad: promedio O(nlogn), peor caso O(n^2)) (pivote muy desfavorable).
- Observación práctica: se puede mitigar el peor caso con pivote aleatorio o mediana de tres.

2.1.2 Merge Sort

- Idea: dividir el arreglo en mitades, ordenar recursivamente cada mitad y fusionar (merge) las listas resultantes.
- Ventaja: garantiza siempre una complejidad O(n log n), es estable y funciona bien con listas grandes.
- Complejidad: O(n log n)
- Observación práctica: Ejemplo práctico de conquista y divide, donde se divide el problema en problemas más pequeños utilizado para grandes volúmenes de datos

2.1.3 Bubble Sort

- Idea: Comparar elementos adyacentes e intercambiarlos si están en el orden incorrecto, haciendo que los elementos más grandes suban al final de la lista en cada pasada.
- Ventaja: Fácil implementación.
- Complejidad: Baja eficiencia en listas grandes. Complejidad $O(n^2)$ en el peor y promedio de los casos.
- Observación práctica: Puede optimizarse terminado el algoritmo antes si en una pasada no se hacen intercambios

2.1.4 InjectionSort

- Idea: Tomar un elemento y colocarlo en su posición correcta dentro de un subarreglo previamente ordenado, repitiendo hasta ordenar todo el conjunto.
- Ventaja: Muy eficiente para listas pequeñas o casi ordenadas; estable.
- Complejidad: Mejor caso O(n) (lista ya ordenada), promedio y peor caso $O(n^2)$.

2.1.5 Swap Sort

- Idea: dividir el arreglo en mitades, ordenar recursivamente y fusionar (merge) las sublistas.
- Ventaja: garantía O(nlogn) en todos los casos y estabilidad.
  Desventaja: uso adicional de memoria O(n) para el merge.

2.2 Algoritmo de ordenamiento en el proyecto

En el proyecto se optó por implementar QuickSort (partición de Lomuto) como método principal de ordenamiento. La elección respondió a varias consideraciones prácticas: QuickSort es simple de implementar recursivamente y suele ofrecer muy buen rendimiento en la práctica (complejidad promedio O(n log n) y buena localidad de memoria, lo que resulta adecuado para vectores en memoria.

Breve descripción técnica del enfoque usado:

- Cada registro se representa con una estructura (`entry`) que contiene: fecha y hora desglosadas, una clave numérica `totalTime` (segundos relativos construidos con la suposición de 31 días por mes), los cuatro octetos de la IP y el puerto, la razón (mensaje) y la línea original.

- El comparador `lessEntry` aplica el criterio de orden deseado: primero `totalTime` (fecha/hora), luego comparación octeto a octeto de la IP (ip1, ip2, ip3, ip4), luego `port` y, como desempate final, la cadena `reason`.

- QuickSort se implementó con partición de Lomuto: se toma el último elemento como pivote, se mueve en una pasada los elementos menores que el pivote hacia la izquierda y se sitúa el pivote en su posición final; luego se recursa en ambos subarreglos.

3. Algoritmos de Búsqueda

3.1 Búsqueda secuencial

- Recorrer y filtrar: O(n) Simple y robusta, pero ineficiente para múltiples consultas sobre datos grandes.

3.2 Búsqueda binaria y límites (lower/upper bound)

- Tras ordenar por la clave temporal, localizar el inicio y final de un rango mediante `lower_bound` (primera posición ≥ claveInicio) y `upper_bound` (primera posición > claveFin) permite obtener resultados en O(logn+k) donde k es el número de registros devueltos. Para rangos temporales esto es ideal.

- Implementar estas búsquedas a mano es sencillo y evita dependencia en librerías si el requisito es hacerlo “desde cero”.

4. Conclusión

El proyecto mostró que una tarea que enunciativamente parece simple leer y ordenar una bitácora exige una buena descomposición del problema .

para lo que fue indispensable:

- Diseñar una estructura de datos clara (`entry`) que agrupe toda la información por registro.
- Implementar funciones auxiliares de parsing (tokenizador, separación de `ip:port`, conversión de hora a clave numérica).
- Elegir y adaptar algoritmos de ordenamiento y búsqueda adecuados al contexto: QuickSort para ordenar con un comparador personalizado, y búsquedas binarias para localizar rangos.

Técnicamente, se alcanzó la funcionalidad requerida: lectura del archivo, ordenamiento por fecha/hora (con desempates por IP/puerto/mensaje), escritura en `sorted.txt` y búsqueda por rango mediante claves numéricas combinadas con `lower`/`upper bound` implementadas manualmente.

5. Reflexión sobre el proyecto de la bitácora

Al inicio de este proyecto me sentía bastante perdido. Aunque tenía una idea general de lo que se buscaba lograr, me faltaba claridad sobre cómo dividir el problema en pasos concretos y programables. El enunciado mencionaba la necesidad de leer, procesar y ordenar una bitácora, pero yo no tenía claro qué estructuras de datos serían más convenientes ni cómo debía organizar mi código para que funcionara de forma robusta.

Uno de los principales retos fue la lectura y el procesamiento de las líneas del archivo. Me encontré con que no era suficiente con leer la cadena completa: había que descomponerla en varias partes (mes, día, hora, IP, puerto y motivo). Ahí fue cuando empecé a comprender la importancia de tener funciones auxiliares claras. Aprendí que usar una función `tokenizer` o un mecanismo para dividir en tokens simplifica mucho el trabajo. Sin embargo, no lo entendí a la primera: batallé bastante con el manejo de la posición en la cadena (`pos`) y con entender qué hacía realmente la función `find` con sus parámetros. En más de una ocasión escribí código que parecía correcto, pero que solo funcionaba en la primera iteración y luego se rompía porque `pos` no avanzaba bien o porque estaba llamando `find` de manera incorrecta.

Otra dificultad fue separar correctamente la IP y el puerto. La primera vez que lo intenté, la función que escribí (`splitIp`) estaba incompleta: no entendía bien cómo usar `find` con un offset y me confundía con los índices. Eso provocó resultados incorrectos que me hacían pensar que el error estaba en otra parte del programa. Este tropiezo me hizo darme cuenta de lo importante que es probar las funciones de forma aislada, con ejemplos concretos,

antes de integrarlas en el resto del proyecto. Con el tiempo logré entender qué `find(".", pos)` busca la siguiente aparición del punto a partir de la posición `pos`, y que si se usa de manera correcta se puede avanzar token por token de forma controlada.

Hubo también un detalle simple que me quitó bastante tiempo: el salto de línea al final del documento. Al principio no entendía por qué, después de procesar todo correctamente, al final aparecía una entrada vacía o algún comportamiento inesperado. La solución era tan simple como saltar líneas vacías justo después de `getline`. Este tipo de errores me enseñó que muchas veces los problemas no están en las partes complicadas, sino en los pequeños detalles que uno pasa por alto.

Una vez que logré leer y descomponer cada registro en su estructura (`entry`), vino el reto de ordenar. Implementar QuickSort no fue tan difícil en lo conceptual, pero sí demandó definir con exactitud el criterio de comparación. Fue necesario construir una clave numérica de tiempo (`totalTime`) para apoyar la comparación cronológica y, adicionalmente, comparar la IP octeto por octeto y el puerto como desempate. Aprendí a diseñar un comparador robusto (`lessEntry`) que cubriera todas las prioridades de orden.

Otro aprendizaje importante fue sobre búsquedas por rango. Inicialmente consideré una búsqueda secuencial para filtrar resultados, pero cuando el conjunto creció entendí que era poco eficiente. Implementar `lowerBoundSum` y `upperBoundSum` me permitió comprender cómo aprovechar la ordenación previa para hacer consultas en O(logn).

Durante el desarrollo hubo momentos de cansancio y falta de ideas; varias veces estuve sin energía y sin saber cómo avanzar. Sin embargo, persistir en pequeñas pruebas, depurar paso a paso y pedir retroalimentación me permitió avanzar. La satisfacción final no fue solo tener el programa que funciona, sino comprender el porqué de cada decisión técnica: modularidad, robustez ante entradas reales y selección de algoritmos adecuados.

6. Referencias

Perez, D. (2025). Algoritmos de ordenamiento [Clase universitaria]. Tec de Moterrey.

C++ structures (struct). (s/f). W3schools.com. Recuperado el 22 de septiembre de 2025, de https://www.w3schools.com/cpp/cpp_structs.asp

(S/f-a). Swhosting.com. Recuperado el 22 de septiembre de 2025, de https://www.swhosting.com/es/comunidad/manual/algoritmos-de-ordenacion-con-ejemplos-en-c#:~:text=Los%20algoritmos%20de%20ordenación%20son,eficiencia%20y%20optimización%20de%20procesos.

(S/f-b). Recuperado el 22 de septiembre de 2025, de http://file:///C:/Users/ismae/Downloads/03%20-%20Algoritmos%20de%20Orenamiento%20Avanzados.pdf

(S/f-c). Recuperado el 22 de septiembre de 2025, de http://file:///C:/Users/ismae/Downloads/02%20-%20Algoritmos%20de%20B%C3%BAsqueda%20y%20Ordenamiento.pdf