Trabajo Integrador de Programación I

“Búsqueda y ordenamiento de datos en catálogo de libros”

Alumnos:

- Clariá Carolina Rocío crclaria@gmail.com
- Diaz Natalia nataliadiazg6@gmail.com

Materia: Programación I

Profesor: AUS Bruselario, Sebastián

Tutor: Candia, Verónica

Fecha de entrega: 09/06/2025.

Índice

1. Introducción

En el ámbito de la informática, las operaciones de ordenamiento y búsqueda constituyen pilares fundamentales para la gestión y procesamiento de colecciones de datos. El ordenamiento se define como el proceso de reorganizar los elementos de un conjunto de datos según un criterio preestablecido, resultando en una disposición sistemática. La búsqueda, por su parte, es el procedimiento de localización de uno o más elementos específicos dentro de una colección, generalmente mediante la comparación de claves o atributos hasta identificar la coincidencia deseada.

Ambas tareas son esenciales porque, al trabajar con volúmenes de información moderados o grandes, el rendimiento de las operaciones que realizamos depende en gran medida de la forma en que organizamos y accedemos a los datos. Una colección ordenada permite acelerar la localización de un elemento concreto.

En este caso práctico se trabajará con un catálogo de 1000 libros, almacenado como una lista de diccionarios en Python, donde cada diccionario incluye al menos los campos "isbn", "titulo" y "autor". El objetivo principal es implementar una función de ordenamiento dinámica que pueda reordenar la lista según cualquiera de esos tres campos, de manera ascendente, utilizando un algoritmo. A continuación, se aplicará la búsqueda binaria sobre la lista ya ordenada para localizar rápidamente un libro a partir de su ISBN, su título o su autor.

De este modo, el caso práctico permitirá comprender cómo representar y manipular colecciones de registros en Python, diseñar una rutina de ordenamiento que reciba como parámetro la clave de orden y devuelva una lista ordenada según esa clave, Implementar la búsqueda binaria sobre la lista ordenada para lograr tiempos de consulta eficientes (O(log n)).

Con este ejercicio se busca reforzar los conceptos de complejidad algorítmica, la relación entre ordenamiento y búsqueda, y la ventaja de escoger la estructura de datos adecuada para obtener soluciones claras, mantenibles y de buen rendimiento.

2.Marco teórico

Algoritmos de Ordenamiento

En la informática, los algoritmos de ordenamiento son cruciales para la optimización de una tarea, estos permiten organizar datos de manera que puedan ser accedidos y utilizados de manera más eficiente. Un algoritmo de ordenamiento permite reorganizar una lista de elementos o nodos en un orden específico, por ejemplo de forma ascendente o descendente dependiendo de la ocasión.

Ordenamientos Simples

- **Burbuja (Bubble Sort)**: es uno de los más simples pero menos eficientes. Funciona comparando pares de elementos e intercambiándolos si están en el orden incorrecto, este proceso se hace una y otra vez hasta que la lista esté ordenada de forma correcta. Este algoritmo tiene una complejidad de tiempo de O(n^2), lo que lo hace útil para ordenar listas pequeñas, pero muy ineficiente para ordenar listas más grandes.

Ventajas:

- Simplicidad: El algoritmo de burbuja es fácil de entender e implementar, lo que lo convierte en una buena opción para introducir conceptos de ordenamiento en la programación.
- Implementación sencilla: Requiere poca cantidad de código y no involucra estructuras de datos complejas.

Desventajas:

- Lento para listas grandes: Debido a su complejidad cuadrática el algoritmo de burbuja se vuelve lento en la práctica para listas de tamaño considerable.

- No considera el orden parcial: A diferencia de otros algoritmos, el algoritmo de burbuja realiza el mismo número de comparaciones e intercambios sin importar si la lista ya está en gran parte ordenada.

- **Inserción (Insertion Sort):** El algoritmo de ordenamiento por inserción es un algoritmo simple pero eficiente. Funciona dividiendo la lista en dos partes, una parte ordenada y otra desordenada, a medida que se recorre la lista desordenada, se insertan elementos en la posición correcta en la parte ordenada.

  El algoritmo de ordenamiento por inserción tiene una complejidad de tiempo de O(n) en el mejor caso y de O(n^2) en el peor caso donde **n** es el número de elementos de la lista.

Ventajas:

- Baja sobrecarga: Requiere menos comparaciones y movimientos que algoritmos como el ordenamiento de burbuja, lo que lo hace más eficiente en términos de intercambios de elementos.
- Simplicidad: el ordenamiento por inserción es uno de los algoritmos de ordenamiento más simples de implementar y entender. Esto lo hace adecuado para enseñar conceptos básicos de ordenamiento.

Desventajas:

- Ineficiencia en listas grandes: A medida que el tamaño de la lista aumenta, el rendimiento del ordenamiento por inserción disminuye. Su complejidad cuadrática de O(n^2) en el pero caso lo hace ineficiente para las listas grandes.
- No escalable: Al igual que otros algoritmos de complejidad cuadrática, el ordenamiento por inserción no es escalable para listas grandes, ya que su tiempo de ejecución aumenta considerablemente con el tamaño de la lista.

- **Selección (Selection Sort)**: El algoritmo de ordenamiento por **Selección** opera bajo el principio de encontrar repetidamente el elemento mínimo (o máximo) de la porción no ordenada de la lista y colocarlo al principio (o al final) de la porción ordenada. Es decir, en cada iteración, busca el elemento más pequeño de la sublista restante y lo intercambia con el primer elemento de esa sublista.

  El algoritmo de seleccion tiene una complejidad de tiempo de O(n2) en todos los casos, porque siempre realiza el mismo número de comparaciones.

Ventajas:

- Simplicidad: Al igual que los anteriores, es fácil de entender e implementar.
- Número mínimo de intercambios: A diferencia de Bubble Sort, Selection Sort realiza un máximo de n−1 intercambios, lo que lo hace útil en situaciones donde los intercambios de elementos son costosos.

Desventajas:

- Ineficiencia para listas grandes: Su complejidad cuadrática lo hace poco práctico para listas de tamaño considerable.
- No adaptable: Su rendimiento no mejora significativamente incluso si la lista está casi ordenada.

Ordenamientos Eficientes

Estos algoritmos tienen una complejidad de tiempo mucho mejor que los simples, lo que los hace adecuados para ordenar grandes volúmenes de datos. Su eficiencia se basa en estrategias más avanzadas, a menudo utilizando recursividad o estructuras de datos complejas.

Merge Sort (Ordenamiento por Mezcla)

Merge Sort es un algoritmo de ordenamiento eficiente que sigue el paradigma "divide y vencerás". Este enfoque implica romper un problema grande en problemas más

pequeños, resolver esos problemas más pequeños, y luego combinar las soluciones para resolver el problema original.

1. Divide: La lista desordenada se divide recursivamente en dos mitades hasta que cada sublista contiene solo un elemento (una lista de un solo elemento se considera ordenada por definición).
2. Vence: Las sublistas de un solo elemento se fusionan ("merged") de forma ordenada.
3. Combina: Las sublistas ordenadas se combinan repetidamente para producir nuevas sublistas ordenadas más grandes, hasta que solo queda una lista, que es la lista original completamente ordenada. El paso de "mezcla" es donde ocurre la mayor parte del trabajo de comparación y ordenamiento.

Tiene una complejidad de tiempo de O(nlogn) en el mejor, promedio y peor caso. Esto se debe a que la división (log n etapas) y la mezcla (n operaciones por etapa) son consistentes.

Ventajas:

- Rendimiento Consistente: Su complejidad de tiempo es siempre O(nlogn), independientemente del estado inicial de la lista, lo que lo hace muy predecible.
- Estable: Es un algoritmo de ordenamiento "estable", lo que significa que mantiene el orden relativo de elementos con valores de clave iguales.
- Ideal para datos externos: Es muy adecuado para ordenar grandes conjuntos de datos que no caben completamente en la memoria principal.

Desventajas:

- Más complejo de implementar: La implementación recursiva puede ser un poco más difícil de entender que los algoritmos simples.

Quick Sort (Ordenamiento Rápido)

Quick Sort es otro algoritmo de ordenamiento eficiente basado en el paradigma "divide y vencerás". Es ampliamente considerado uno de los algoritmos de ordenamiento más rápidos en la práctica para una gran variedad de datos, aunque su complejidad en el peor caso es cuadrática.

1. Divide: El algoritmo selecciona un elemento de la lista llamado pivote. Luego, reorganiza los demás elementos de la lista de tal manera que todos los elementos menores que el pivote queden antes que él, y todos los elementos mayores queden después. Este proceso se llama particionamiento.
2. Vence: El algoritmo se aplica recursivamente a las sublistas de elementos menores y mayores al pivote.
3. Combina: (Implícito) A diferencia de Merge Sort, no hay un paso explícito de "combinación"; la lista se ordena "en su lugar" a medida que se realizan las particiones.

Tiene una complejidad de tiempo de O(nlogn) en el mejor y promedio caso y de O(n2) en el peor caso (cuando la elección del pivote es consistentemente mala, como elegir siempre el elemento más pequeño o más grande, lo que resulta en particiones desequilibradas).

Ventajas:

- Alto rendimiento en la práctica: Generalmente, es más rápido que otros algoritmos O(nlogn) para la mayoría de los casos.

Desventajas:

- Rendimiento en el peor caso: Si el pivote se elige mal repetidamente, su rendimiento puede degradarse a O(n2), lo que lo hace comparable a los algoritmos simples.
- No es estable: No garantiza el orden relativo de elementos con valores de clave iguales.

Heap Sort (Ordenamiento por Montículo)

Heap Sort es un algoritmo de ordenamiento basado en la estructura de datos heap (montículo). Un heap es un árbol binario completo que satisface la propiedad de heap: para un max-heap, el valor de cada nodo es mayor o igual que el valor de sus hijos; para un min-heap, es menor o igual. Heap Sort utiliza un max-heap.

1. Construir el Heap: La lista se transforma en un max-heap.
2. Extraer elementos: El elemento más grande (la raíz del heap) se extrae y se coloca al final de la lista. Luego, el heap se "restaura" para mantener la propiedad de heap. Este proceso se repite n veces hasta que todos los elementos han sido extraídos y la lista está ordenada.

Tiene una complejidad de tiempo de O(nlogn) en todos los casos (mejor, promedio y peor).

Ventajas:

- Rendimiento consistente: Al igual que Merge Sort, su complejidad de tiempo es siempre O(nlogn).

Desventajas:

- No es estable: No mantiene el orden relativo de elementos con valores de clave iguales.
- Menos eficiente en la práctica: A pesar de su buena complejidad teórica, en la práctica suele ser un poco más lento que Quick Sort debido a patrones de acceso a la memoria menos favorables.

Algoritmos de Búsqueda

Los algoritmos de búsqueda son métodos que nos permiten encontrar la ubicación de un elemento específico dentro de una lista de elementos. Dependiendo de la lista necesitarás utilizar un algoritmo u otro, por ejemplo si la lista tiene elementos ordenados, puedes usar un algoritmo de búsqueda binaria, pero si la lista contiene los

elementos de forma desordenada este algoritmo no te servirá, para búscar un elemento en una lista desordenada deberás utilizar un algoritmo de búsqueda lineal. Estos algoritmos son dos de los más relevantes y conocidos en la programación, a continuación veremos ejemplos de estos dos algoritmos.

Búsqueda Lineal

Los algoritmos de búsqueda lineal, también conocidos como búsqueda secuencial, implican recorrer una lista de elementos uno por uno hasta encontrar un elemento específico. Este algoritmo es muy sencillo de implementar en código pero puede ser muy ineficiente dependiendo del largo de la lista y la ubicación donde está el elemento.

Ventajas:

- Sencillez: La búsqueda lineal es uno de los algoritmos de búsqueda más simples y fáciles de implementar. Solo requiere iterar a través de la lista de elementos uno por uno hasta encontrar el objetivo.
- flexibilidad: La búsqueda lineal puede aplicarse a cualquier tipo de lista, independientemente de si está ordenada o no.

Desventajas:

- Ineficiencia en listas grandes: La principal desventaja de la búsqueda lineal es su ineficiencia en listas grandes. Debido a que compara cada elemento uno por uno, su tiempo de ejecución crece de manera lineal con el tamaño de la lista.

Búsqueda Binaria

- El algoritmo de búsqueda binaria es un algoritmo muy eficiente que se aplica solo a listas ordenadas. Funciona dividiendo repetidamente la lista en dos mitades y comparando el elemento objetivo con el elemento del medio, esto reduce significativamente la cantidad de comparaciones necesarias.

Ventajas:

- Eficiencia de listas ordenadas: La principal ventaja de la búsqueda binaria es su eficiencia en listas ordenadas. Su tiempo de ejecución es de O(log n), lo que significa que disminuye rápidamente a medida que el tamaño de la lista aumenta.
- Menos comparaciones: Comparado con la búsqueda lineal, la búsqueda binaria realiza menos comparaciones en promedio, lo que lo hace más rápido para encontrar el objetivo.

Desventajas:

- Requiere una lista ordenada: La búsqueda binaria sólo es aplicable a listas ordenadas, Si la lista no está ordenada, se debe realizar una operación adicional para ordenarla antes de usar la búsqueda binaria.
- Mayor complejidad de implementación: Comparado con la búsqueda lineal, la búsqueda binaria es más compleja de implementar debido a su naturaleza recursiva.

3. Caso práctico

En el presente caso práctico se implementará un sistema de ordenamiento y búsqueda binaria para un catálogo de 30 libros, donde cada libro estará representado como un diccionario en Python. Este enfoque permite almacenar, de manera estructurada, atributos clave de cada obra literaria (por ejemplo: isbn, título y autor), facilitando el acceso y la manipulación de los datos. El uso de diccionarios proporciona un mapeo directo entre la clave de búsqueda (como el código ISBN) y el resto de la información asociada a cada libro, mejorando la legibilidad y la mantenibilidad del código.

El objetivo principal de este ejercicio es demostrar cómo una colección de registros puede someterse a un proceso de ordenamiento basado en diferentes atributos, y posteriormente ejecutar una búsqueda binaria sobre la clave correspondiente. Para que la búsqueda binaria sea válida, la lista de diccionarios debe estar previamente ordenada por la clave escogida. De esta manera, se garantiza una complejidad de búsqueda de O(log n) sobre el catálogo completo, lo cual resulta especialmente beneficioso cuando se trabaja con volúmenes de datos moderados a grandes.

Se realizará en dos pasos principales:

1. **Proceso de ordenamiento**: Se implementará una función genérica que reciba como parámetro el campo por el cual se desea ordenar (por ejemplo, key="titulo") y que, mediante un algoritmo eficiente, genere una lista ordenada de manera ascendente.

2. **Búsqueda binaria**: Una vez que el catálogo esté ordenado por un campo específico, se aplicará la búsqueda binaria para localizar rápidamente un libro dado su ISBN, su título exacto o su autor. El algoritmo comparará, de forma recursiva, el valor medio de la lista con la clave buscada, reduciendo sistemáticamente el rango de búsqueda hasta hallar la coincidencia o determinar que no existe.

4. Metodología Utilizada

Proceso de ordenamiento:

En esta sección del proyecto, el objetivo principal fue desarrollar un sistema para organizar y gestionar una lista de libros. Para lograrlo, nos enfocamos en la implementación de un algoritmo de ordenamiento que permitiera clasificar los libros según su título, autor, año o número de ISBN.

Como punto de partida para este caso práctico, la lista de libros utilizada fue proporcionada por una inteligencia artificial (IA), sirviendo como el conjunto de datos inicial para aplicar y probar los algoritmos de ordenamiento y búsqueda desarrollados.

El algoritmo elegido es Merge Sort (Ordenamiento por Mezcla). Este se implementó a través de dos funciones principales: (las capturas de pantalla de ejecución del código se anexaron a la parte 8.Anexos)

1. merge_sort_libros(lista, key): Esta función es el punto de entrada del algoritmo. Sigue el principio "divide y vencerás". Su tarea es tomar una lista de libros (que son diccionarios, cada uno con claves como "titulo", "autor", "anio", "isbn") y una key específica por la cual se desea ordenar (por ejemplo, "titulo"). Recursivamente, divide la lista en mitades más pequeñas hasta que cada sublista contiene solo un elemento. Luego, invoca a la función merge para combinar estas sublistas de forma ordenada.

```
def merge_sort_libros(libros, key):
    if len(libros) <= 1:
        return libros
    # Dividir la lista en dos mitades
    mid = len(libros) // 2
    mitad_izq = libros[:mid]
    mitad_der = libros[mid:]
    # Llamada recursiva para ordenar las mitades
    mitad_izq = merge_sort_libros(mitad_izq, key)
    mitad_der = merge_sort_libros(mitad_der, key)
    # Combinar las mitades ordenadas
    return merge(mitad_izq, mitad_der, key)
```

2. merge(lista_izquierda, lista_derecha, key): Esta función auxiliar es donde ocurre la lógica de fusión y ordenamiento. Recibe dos sublistas ya ordenadas y las combina en una única lista ordenada. Compara los elementos de ambas sublistas basándose en el valor de la key especificada. Es importante destacar que, para asegurar una comparación precisa e insensible a mayúsculas/minúsculas (en el caso de cadenas de texto como títulos o autores) y para manejar posibles espacios en blanco, los valores de la key son convertidos a minúsculas y se les eliminan los espacios (.lower().strip()) antes de cada comparación. Esto garantiza que "El Quijote" y "el quijote " se ordenen correctamente.

```
def merge(izq, der, key):
    merged_lista = []
    i = 0  # Índice para la lista izquierda
    j = 0  # Índice para la lista derecha
    while i < len(izq) and j < len(der):
        if izq[i][key] <= der[j][key]:
            merged_lista.append(izq[i])
            i += 1
        else:
            merged_lista.append(der[j])
            j += 1
    # Agregar los elementos restantes de cualquiera de las listas
    while i < len(izq):
        merged_lista.append(izq[i])
        i += 1
    while j < len(der):
        merged_lista.append(der[j])
        j += 1
    return merged_lista
```

La implementación de estas funciones nos permite tomar nuestra lista inicial de libros y obtener nuevas listas ordenadas por cualquier campo deseado, como el título, el año, el autor o el ISBN. Por ejemplo, al ordenar por "autor", todos los libros de "Gabriel García Márquez" aparecerán juntos y en el orden lexicográfico correcto. Este paso de

ordenamiento es crítico, ya que la eficiencia de la posterior búsqueda binaria depende directamente de que los datos estén previamente ordenados.

La elección de **Merge Sort** para el desarrollo de la parte de ordenamiento de este trabajo práctico se basa en las siguientes consideraciones clave:

1. Consistencia de Rendimiento: Merge Sort ofrece una complejidad de tiempo de O(nlogn) en todos los casos (mejor, promedio y peor). A diferencia de Quick Sort, que puede degradarse a O(n2) en el peor caso (con una mala elección de pivote), Merge Sort garantiza un rendimiento predecible y eficiente. Para un sistema que procesa datos de manera constante, la predictibilidad en el tiempo de ejecución es una ventaja significativa.
2. Estabilidad: Merge Sort es un algoritmo de ordenamiento estable. Esto significa que si dos elementos tienen el mismo valor en la clave de ordenamiento (por ejemplo, dos libros con el mismo autor), su orden relativo original se mantiene después de la ordenación. Esta propiedad es crucial en aplicaciones donde el orden secundario de los datos es importante.
3. Facilidad de Implementación Recursiva (y Comprensión del Paradigma Divide y Vencerás): Si bien puede parecer más complejo que los ordenamientos simples, Merge Sort ilustra de manera elegante y clara el poderoso paradigma "Divide y Vencerás" de la programación. Su naturaleza recursiva es intuitiva una vez que se comprende el concepto de dividir y luego fusionar las sub listas ordenadas.
4. Adecuado para Listas de Objetos Complejos: En nuestro caso, estamos ordenando una lista de libros basándonos en una clave específica ("titulo", "autor", "isbn", “año”). Merge Sort maneja esto muy bien porque la comparación se realiza sobre el valor de la clave, y el proceso de mezcla se encarga de reensamblar los diccionarios completos, garantizando que los objetos se muevan de forma íntegra.

Proceso de búsqueda

Una vez ordenada la lista de libros mediante el algoritmo Merge Sort, el siguiente paso metodológico del proyecto fue desarrollar un sistema de búsqueda que permitiera encontrar libros específicos de forma eficiente, utilizando criterios como título, autor, año de publicación o número de ISBN.

El algoritmo elegido fue búsqueda binaria, que requiere que la lista está ordenada. Se implementó a través de una función:

```
def busqueda_binaria(lista, key, valor_busqueda):
    # Convertimos el valor de búsqueda a minúsculas .lower() y sin espacios
                        .strip()para la comparación.
    valor_busqueda_lower = str(valor_busqueda).lower().strip()
    if len(lista) == 0:
        return "Valor no encontrado"
    medio = len(lista) // 2
    valor_central = lista[medio][key]
    # Convertimos el valor central del diccionario a minúsculas y sin espacios.
    valor_central_lower = str(valor_central).lower().strip()

    if valor_central_lower == valor_busqueda_lower:
        # Si lo encontramos, devolvemos el diccionario completo (no solo
                        imprimir)
        # Esto hace la función más reutilizable.
        return lista[medio]
    else:
        if valor_busqueda_lower < valor_central_lower:
            return busqueda_binaria(lista[:medio], key, valor_busqueda)
        else:
            return busqueda_binaria(lista[medio + 1:], key, valor_busqueda)
```

Esta función recibe tres parámetros:

- lista: la lista de libros ya ordenada según la clave;
- key: la clave por la cual se desea buscar (por ejemplo, "titulo");
- valor_busqueda: el valor exacto que se quiere encontrar.

El proceso consiste en comparar el valor buscado con el elemento central de la lista. Si hay coincidencia, se devuelve el diccionario correspondiente al libro encontrado. En caso contrario, la búsqueda continúa de forma recursiva en la mitad izquierda o derecha de la lista, dependiendo del resultado de la comparación.

Para garantizar comparaciones precisas, tanto el valor buscado como el valor central son transformados a minúsculas y se les eliminan espacios innecesarios (.lower().strip()), evitando errores causados por diferencias de formato.

La decisión de utilizar búsqueda binaria se fundamenta en los siguientes aspectos:

- Eficiencia: La búsqueda binaria tiene una complejidad temporal de O(log n), significativamente más rápida que una búsqueda secuencial (O(n)), especialmente en listas grandes.
- Reutilización del ordenamiento previo: Al haber ordenado previamente la lista con Merge Sort, se cumplen las condiciones necesarias para que la búsqueda binaria funcione correctamente.
- Aplicación a datos estructurados: Al trabajar con una lista de diccionarios, la búsqueda binaria fue adaptada para comparar únicamente sobre el campo relevante (key), manteniendo la integridad del resto del objeto (libro).

5. Resultados Obtenidos

La implementación de Merge Sort resultó en un sistema de ordenamiento robusto y funcional, capaz de organizar el catálogo de libros eficientemente por cualquiera de los campos definidos.

- Catálogos Ordenados: Se generaron exitosamente listas de libros ordenadas por Título, Autor, ISBN y Año de publicación.
- Insensibilidad a Mayúsculas/Minúsculas y Espacios: Gracias a la normalización de las claves (.lower().strip()) dentro de la función merge, el ordenamiento es consistente y no se ve afectado por variaciones en el uso de mayúsculas/minúsculas o espacios adicionales en los datos de entrada, lo que mejora la calidad y fiabilidad del ordenamiento.
- Base para Búsquedas Eficientes: La principal ventaja de tener el catálogo ordenado es que sienta las bases para implementar algoritmos de búsqueda altamente eficientes, como la búsqueda binaria. Al garantizar que los datos están en un orden predecible, operaciones posteriores como la localización de un libro específico se realizan en tiempo logarítmico (O(logn)), lo que es fundamental para catálogos de gran tamaño.

La implementación de la búsqueda binaria recursiva resultó en un método eficiente y directo para localizar libros dentro de un catálogo previamente ordenado.

Gracias a la naturaleza del algoritmo binario, la búsqueda opera con una complejidad temporal de O(log n), lo que permite localizar elementos con gran rapidez incluso en listas extensas. Este comportamiento logarítmico es posible solo si la lista ya se encuentra ordenada por la clave de búsqueda, lo cual fue garantizado por la etapa previa de ordenamiento con Merge Sort.

Una limitación importante de esta implementación es que solo devuelve el primer elemento que coincide exactamente con el valor de búsqueda. En casos donde múltiples libros comparten el mismo valor en la clave seleccionada (por ejemplo, varios libros del

mismo autor o publicados el mismo año), la función retorna únicamente una de esas entradas, omitiendo las demás. Esto restringe la funcionalidad en contextos donde el usuario espera recuperar todas las coincidencias posibles.

6. Conclusiones

Este trabajo práctico permitió una comprensión sólida y una aplicación concreta de algoritmos fundamentales en la informática, en particular el algoritmo de ordenamiento Merge Sort y la búsqueda binaria simple. La elección de Merge Sort resultó especialmente acertada debido a su rendimiento predecible (O(n log n) en todos los casos), su estabilidad (preservando el orden relativo de elementos con claves iguales), y su claridad conceptual al implementar el paradigma "divide y vencerás".

La implementación de Merge Sort resolvió eficazmente la necesidad de ordenar un catálogo de libros en base a distintos criterios (título, autor, año, ISBN), permitiendo que los datos estén estructurados de forma coherente. Un aspecto destacado fue la incorporación de técnicas de normalización de datos (conversión a minúsculas y eliminación de espacios innecesarios), lo cual aseguró comparaciones consistentes y precisas, tanto en el proceso de ordenamiento como en la búsqueda posterior.

Complementando este ordenamiento, se desarrolló una función de búsqueda binaria, que permitió localizar libros de forma eficiente, aprovechando el orden establecido previamente. Este algoritmo ofreció un acceso rápido a la información con complejidad logarítmica (O(log n)), una mejora significativa en comparación con la búsqueda secuencial.

Sin embargo, se identificó una limitación importante: la búsqueda binaria implementada fue diseñada para devolver únicamente el primer resultado coincidente. En contextos donde múltiples libros comparten el mismo valor en la clave de búsqueda (por ejemplo, varios libros del mismo autor), esta función devuelve solo uno de ellos, lo cual acota su aplicabilidad en esos escenarios.

A pesar de esta limitación, la solución desarrollada constituye una base sólida para la gestión eficiente de colecciones de datos. Sienta también las condiciones para futuras mejoras, como la búsqueda de múltiples coincidencias o la incorporación de filtros compuestos, extendiendo la utilidad del sistema a contextos más complejos.

7. Bibliografía

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. (Conocido popularmente como CLRS)

- Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley Professional.

- Python Oficial: https://docs.python.org/3/library/

- OpenAI. (2025). Asistencia virtual a través de ChatGPT. Datos generados bajo solicitud para el "Catálogo de Libros".
- Videos tutoriales: https://www.youtube.com/playlist?list=PLn3VjQYU7QJaDj3eGZhJLsurag29ps5VE
- Universidad Tecnológica Nacional (UTN). Material de la cátedra “Programación 1” – Unidad: Búsqueda, ordenamiento y análisis de algoritmos. Tecnicatura Universitaria en Programación a Distancia, UTN.

8. Anexos

-Capturas de pantalla del programa funcionando:

```
Catálogo ordenado por título:
  Título: 1984, Autor: George Orwell
  Título: A sangre fría, Autor: Truman Capote
  Título: Cien años de soledad, Autor: Gabriel García Márquez
  Título: Crimen y castigo, Autor: Fiódor Dostoyevski
  Título: Don Quijote de la Mancha, Autor: Miguel de Cervantes
  Título: Drácula, Autor: Bram Stoker
  Título: El Hobbit, Autor: J.R.R. Tolkien
  Título: El alquimista, Autor: Paulo Coelho
  Título: El código Da Vinci, Autor: Dan Brown
  Título: El gran Gatsby, Autor: F. Scott Fitzgerald
  Título: El guardián entre el centeno, Autor: J.D. Salinger
  Título: El principito, Autor: Antoine de Saint-Exupéry
  Título: El retrato de Dorian Gray, Autor: Oscar Wilde
  Título: El señor de los anillos, Autor: J.R.R. Tolkien
  Título: Ensayo sobre la ceguera, Autor: José Saramago
  Título: Fahrenheit 451, Autor: Ray Bradbury
  Título: Harry Potter y la piedra filosofal, Autor: J.K. Rowling
  Título: It, Autor: Stephen King
  Título: La carretera, Autor: Cormac McCarthy
  Título: La casa de los espíritus, Autor: Isabel Allende
  Título: La ladrona de libros, Autor: Markus Zusak
  Título: La sombra del viento, Autor: Carlos Ruiz Zafón
  Título: Las aventuras de Huckleberry Finn, Autor: Mark Twain
  Título: Lolita, Autor: Vladimir Nabokov
  Título: Los juegos del hambre, Autor: Suzanne Collins
  Título: Los miserables, Autor: Victor Hugo
  Título: Matar a un ruiseñor, Autor: Harper Lee
  Título: Orgullo y prejuicio, Autor: Jane Austen
  Título: Rayuela, Autor: Julio Cortázar
  Título: Ulises, Autor: James Joyce
```

Catálogo ordenado por título.

```
Catálogo ordenado por autor:
  Autor: Antoine de Saint-Exupéry, Título: El principito
  Autor: Bram Stoker, Título: Drácula
  Autor: Carlos Ruiz Zafón, Título: La sombra del viento
  Autor: Cormac McCarthy, Título: La carretera
  Autor: Dan Brown, Título: El código Da Vinci
  Autor: F. Scott Fitzgerald, Título: El gran Gatsby
  Autor: Fiódor Dostoyevski, Título: Crimen y castigo
  Autor: Gabriel García Márquez, Título: Cien años de soledad
  Autor: George Orwell, Título: 1984
  Autor: Harper Lee, Título: Matar a un ruiseñor
  Autor: Isabel Allende, Título: La casa de los espíritus
  Autor: J.D. Salinger, Título: El guardián entre el centeno
  Autor: J.K. Rowling, Título: Harry Potter y la piedra filosofal
  Autor: J.R.R. Tolkien, Título: El señor de los anillos
  Autor: J.R.R. Tolkien, Título: El Hobbit
  Autor: James Joyce, Título: Ulises
  Autor: Jane Austen, Título: Orgullo y prejuicio
  Autor: José Saramago, Título: Ensayo sobre la ceguera
  Autor: Julio Cortázar, Título: Rayuela
  Autor: Mark Twain, Título: Las aventuras de Huckleberry Finn
  Autor: Markus Zusak, Título: La ladrona de libros
  Autor: Miguel de Cervantes, Título: Don Quijote de la Mancha
  Autor: Oscar Wilde, Título: El retrato de Dorian Gray
  Autor: Paulo Coelho, Título: El alquimista
  Autor: Ray Bradbury, Título: Fahrenheit 451
  Autor: Stephen King, Título: It
  Autor: Suzanne Collins, Título: Los juegos del hambre
  Autor: Truman Capote, Título: A sangre fría
  Autor: Victor Hugo, Título: Los miserables
  Autor: Vladimir Nabokov, Título: Lolita
```

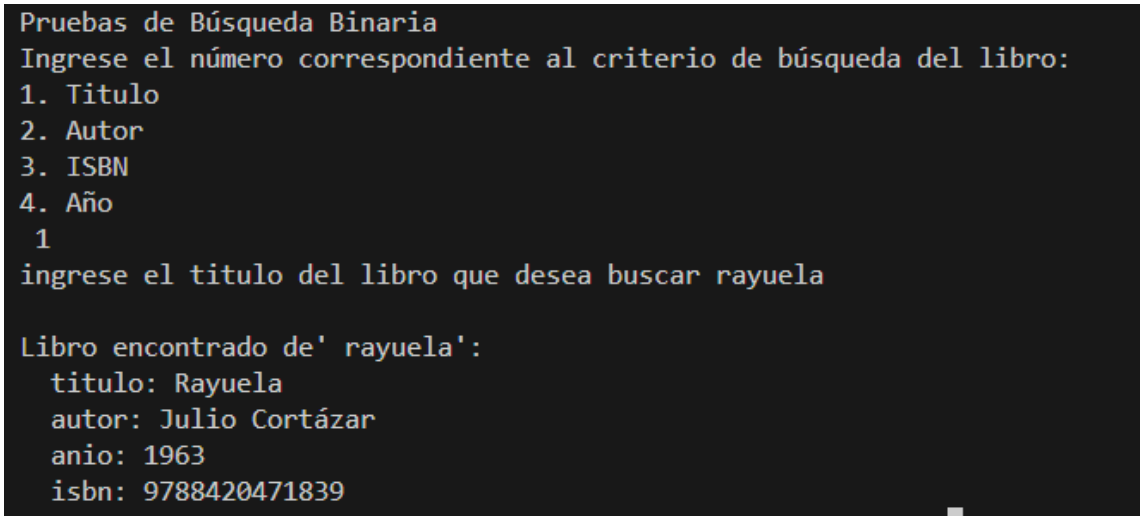
Catálogo ordenado por autor.

```
Catálogo ordenado por ISBN:
  ISBN: 9780061120084, Título: Matar a un ruiseñor
  ISBN: 9780061122415, Título: El alquimista
  ISBN: 9780140449136, Título: Crimen y castigo
  ISBN: 9780141182803, Título: Ulises
  ISBN: 9780141439518, Título: Orgullo y prejuicio
  ISBN: 9780141439570, Título: El retrato de Dorian Gray
  ISBN: 9780141439846, Título: Drácula
  ISBN: 9780143107323, Título: Las aventuras de Huckleberry Finn
  ISBN: 9780156007757, Título: Ensayo sobre la ceguera
  ISBN: 9780156012195, Título: El principito
  ISBN: 9780307387899, Título: La carretera
  ISBN: 9780307474278, Título: El código Da Vinci
  ISBN: 9780307474728, Título: Cien años de soledad
  ISBN: 9780316769488, Título: El guardián entre el centeno
  ISBN: 9780345339683, Título: El Hobbit
  ISBN: 9780375842207, Título: La ladrona de libros
  ISBN: 9780439023528, Título: Los juegos del hambre
  ISBN: 9780451419439, Título: Los miserables
  ISBN: 9780451524935, Título: 1984
  ISBN: 9780544003415, Título: El señor de los anillos
  ISBN: 9780553383805, Título: La casa de los espíritus
  ISBN: 9780679723165, Título: Lolita
  ISBN: 9780679745587, Título: A sangre fría
  ISBN: 9780743273565, Título: El gran Gatsby
  ISBN: 9781451673319, Título: Fahrenheit 451
  ISBN: 9781501142970, Título: It
  ISBN: 9788408172173, Título: La sombra del viento
  ISBN: 9788420471839, Título: Rayuela
  ISBN: 9788478884452, Título: Harry Potter y la piedra filosofal
  ISBN: 9788491050291, Título: Don Quijote de la Mancha
```

Catálogo ordenado por número ISBN.

```
Catálogo ordenado por año:
  Año: 1605, Título: Don Quijote de la Mancha
  Año: 1813, Título: Orgullo y prejuicio
  Año: 1862, Título: Los miserables
  Año: 1866, Título: Crimen y castigo
  Año: 1884, Título: Las aventuras de Huckleberry Finn
  Año: 1890, Título: El retrato de Dorian Gray
  Año: 1897, Título: Drácula
  Año: 1922, Título: Ulises
  Año: 1925, Título: El gran Gatsby
  Año: 1937, Título: El Hobbit
  Año: 1943, Título: El principito
  Año: 1949, Título: 1984
  Año: 1951, Título: El guardián entre el centeno
  Año: 1953, Título: Fahrenheit 451
  Año: 1954, Título: El señor de los anillos
  Año: 1955, Título: Lolita
  Año: 1960, Título: Matar a un ruiseñor
  Año: 1963, Título: Rayuela
  Año: 1966, Título: A sangre fría
  Año: 1967, Título: Cien años de soledad
  Año: 1982, Título: La casa de los espíritus
  Año: 1986, Título: It
  Año: 1988, Título: El alquimista
  Año: 1995, Título: Ensayo sobre la ceguera
  Año: 1997, Título: Harry Potter y la piedra filosofal
  Año: 2001, Título: La sombra del viento
  Año: 2003, Título: El código Da Vinci
  Año: 2005, Título: La ladrona de libros
  Año: 2006, Título: La carretera
  Año: 2008, Título: Los juegos del hambre
```

Catálogo ordenado por año de publicación.



Búsqueda binaria por titulo

-Repositorio en GitHub:

https://github.com/caroclaria/integrador-programacion-I

-Video explicativo: (insertar enlace al video en YouTube)

https://youtu.be/Zp3xdbmHLIs