UNIVERSIDAD DE CALDAS

Facultad de Ingenierías

Informe Técnico:

Sistema de Gestión de Bibliotecas (SGB)

Elaborado por:

Juan Camilo Cruz Parra

Jaime Andrés Cardona Díaz

Daner Alejandro Salazar Colorado

Presentado a:

Mario Alejandro Bravo Ortiz

Asignatura:

Técnicas de Programación

Manizales, Caldas

2025

Índice general

1. Sistema de Gestión de Bibliotecas (SGB)

1.1 Descripción General del Sistema

El **Sistema de Gestión de Bibliotecas (SGB)** es una aplicación de software diseñada para administrar el flujo de inventario, préstamos y usuarios de una biblioteca. El sistema está construido bajo una arquitectura **modular por capas**, separando claramente la lógica de negocio, las estructuras de datos y los algoritmos utilitarios.

El núcleo del sistema permite:

- Carga masiva de libros desde archivos (CSV/JSON).
- Gestión de usuarios y control de préstamos.
- Búsqueda eficiente de material bibliográfico.
- Generación de reportes ordenados.
- Resolución de problemas de optimización (estantería) y cálculos estadísticos recursivos.

La interacción principal se realiza a través de una interfaz de consola (`Main.py`), que actúa como punto de entrada y orquestador de las peticiones del usuario hacia el controlador principal (`Biblioteca.py`).

1.2 Estructuras de Datos Utilizadas

El sistema implementa estructuras de datos tanto nativas como personalizadas para manejar la información de manera eficiente.

1.2.1 Listas y Diccionarios (Nativas)

- **Inventario General (`List[Libro]`):** almacena todos los libros en orden de llegada. Permite una complejidad de inserción $O(1)$.
- **Inventario Ordenado (`List[Libro]`):** lista paralela mantenida en orden ascendente por ISBN. Es fundamental para habilitar la búsqueda binaria.
- **Usuarios (`Dict[str, Usuario]`):** mapa hash que permite acceso prácticamente $O(1)$ a los usuarios mediante su ID.

1.2.2 Pilas (LIFO - Last In, First Out)

Se implementó una clase `Pila` personalizada para gestionar el **historial de préstamos** de cada usuario.

Justificación: el historial de un usuario se consulta generalmente buscando "lo último que leyó". Una pila es la estructura natural para esto, permitiendo acceso $O(1)$ al préstamo más reciente sin necesidad de recorrer todo el historial.

```
class Pila:
    def __init__(self):
        self.items = []

    def apilar(self, item):
        self.items.append(item)

    def desapilar(self):
        if not self.esta_vacia():
            return self.items.pop()
```

Listing 1.1: Fragmento de DataStructures/Pila.py

1.2.3 Estructura de Espera (Gestión de Reservas)

Para manejar libros sin stock, se utiliza la clase `PilaDeEspera`. Aunque conceptualmente una cola (FIFO) sería lo estándar para turnos, el sistema implementa una lógica de espera donde las solicitudes se apilan.

Funcionamiento: cuando un libro es devuelto, el sistema verifica automáticamente esta estructura para asignar el libro al siguiente usuario en espera según las reglas definidas.

1.3 Algoritmos Implementados

El proyecto destaca por la implementación manual de algoritmos clásicos de computación, evitando el uso excesivo de funciones "caja negra" de librerías externas.

1.3.1 Algoritmos de Ordenamiento

Merge Sort (Ordenamiento por Mezcla)

Utilizado para generar el **Reporte de Inventario por Valor**.

- **Complejidad:** $O(n \log n)$ en todos los casos.
- **Justificación:** se eligió por su **estabilidad** (mantiene el orden relativo de elementos iguales) y su eficiencia garantizada incluso con grandes volúmenes de datos, a diferencia de Quicksort que puede degradarse a $O(n^2)$.

```
def merge_sort_por_valor(libros):
```

```
    if len(libros) <= 1:
        return libros
    mid = len(libros) // 2
    left = merge_sort_por_valor(libros[:mid])
    right = merge_sort_por_valor(libros[mid:])
    return _merge_valor(left, right)
```

Listing 1.2: Merge sort por valor

Insertion Sort (Ordenamiento por Inserción)

Utilizado para ordenar el **historial de préstamos por fecha**.

- **Complejidad:** $O(n^2)$ en promedio, pero $O(n)$ en listas casi ordenadas.
- **Justificación:** los historiales de préstamos suelen ser pequeños y se construyen cronológicamente. Insertion Sort es extremadamente rápido y eficiente en memoria para estos casos de datos casi ordenados.

1.3.2 Algoritmos de Búsqueda

Búsqueda Binaria

Utilizada para buscar libros por **ISBN**.

- **Requisito:** la lista debe estar ordenada (se usa `inventario_ordenado_isbn`).
- **Complejidad:** $O(\log n)$.
- **Impacto:** permite encontrar un libro entre un millón de registros en aproximadamente 20 comparaciones, frente a un millón de comparaciones con búsqueda lineal.

```
def busqueda_binaria_isbn(inventario, isbn):
    low = 0
    high = len(inventario) - 1
    while low <= high:
        mid = (low + high) // 2
        if inventario[mid].isbn == isbn:
            return mid
        elif inventario[mid].isbn < isbn:
            low = mid + 1
        else:
            high = mid - 1
    return None
```

Listing 1.3: Búsqueda binaria por ISBN

Búsqueda Lineal

Utilizada para buscar por **título** o **autor**.

- **Complejidad:** $O(n)$.
- **Justificación:** necesaria porque no podemos mantener el inventario ordenado por todos los criterios simultáneamente. Además, permite coincidencias parciales (subcadenas).

1.3.3 Recursividad

Recursión de Pila (Stack Recursion)

Implementada, por ejemplo, para encontrar el **libro más ligero** de un autor. La función se llama a sí misma procesando el resto de la lista y compara el resultado con el elemento actual al "volver" de la recursión.

Recursión de Cola (Tail Recursion)

Implementada para calcular el **peso promedio** de los libros de un autor.

- **Lógica:** utiliza acumuladores en los parámetros de la función. La llamada recursiva es la última instrucción, lo que teóricamente permite a compiladores/intérpretes optimizar el uso de memoria (*Tail Call Optimization*).

1.3.4 Backtracking (Optimización Combinatoria)

Utilizado en el **módulo de estantería** para seleccionar los mejores libros bajo una restricción de peso (problema de la mochila, *Knapsack Problem*).

- **Estrategia:** exploración en profundidad del árbol de decisiones.
- **Poda (*pruning*):** el algoritmo descarta inmediatamente ramas que no llevan a una solución válida (por ejemplo, libros con peso menor a 0.5 kg o combinaciones que exceden el peso máximo acumulado), reduciendo drásticamente el tiempo de ejecución.

```
if libro.peso < 0.5:
    continue  # PODA: Ignorar libros muy ligeros
if peso_actual + libro.peso <= max_peso:
```

```
    # Explorar rama incluyendo el libro...
    ...
```

Listing 1.4: Poda básica en módulo de estantería

1.4 Resultados de Ejecución (Simulación)

A continuación se presentan resultados representativos basados en las pruebas del sistema.

1.4.1 Caso 1: Búsqueda y Préstamo

Entrada: usuario `Ü001"` solicita libro ISBN `"978-4"` (IA Moderna).

Proceso:

1. `busqueda_binaria_isbn` localiza `"978-4"` en muy pocos pasos.
2. El sistema verifica el stock. Si `stock >0`, crea un objeto `Prestamo` y lo apila en `U001.historial`.

Salida: `Préstamo exitoso: 'IA Moderna' entregado a Mario Bravo.`

1.4.2 Caso 2: Optimización de Estantería

Entrada: inventario con 5 libros, peso máximo = 8.0 kg.

Proceso: el algoritmo de backtracking explora combinaciones:

- Descarta *El Principito* (0.2 kg) por la regla de poda (peso menor a 0.5 kg).
- Evalúa combinaciones restantes para maximizar el valor en pesos sin superar los 8 kg.

Salida (ejemplo):

```
--- MEJOR SELECCIÓN ENCONTRADA ---
Valor Total: $420,000
Peso Total: 4.50 kg
Libros:
 - IA Moderna ($250,000, 2.5kg)
 - Clean Code ($180,000, 0.8kg)
 - Estructuras de Datos ($120,000, 1.2kg)
```

1.5 Justificación de Decisiones Técnicas

1. **Dualidad de inventarios:** se decidió mantener dos listas (`general` y `ordenada`) sacrificando memoria RAM para ganar velocidad de CPU. Esto permite inserciones rápidas ($O(1)$ en la general) y búsquedas rápidas ($O(\log n)$ en la ordenada) simultáneamente.

2. **Uso de pila para historial:** modelar el historial como una pila es una decisión de diseño centrada en el usuario. En sistemas de recomendación y consulta, la actividad reciente tiene mucho más peso que la antigua.
3. **Backtracking vs. fuerza bruta:** aunque se pueden comparar ambos enfoques, el backtracking con poda es la solución práctica. La fuerza bruta (evaluar todas las combinaciones) es computacionalmente inviable ($O(2^n)$) a medida que crece el inventario. La poda reduce este espacio de búsqueda significativamente.
4. **Modularidad:** la separación en carpetas (`Controller`, `Entities`, `UtilityAlgorithms`) permite que múltiples desarrolladores trabajen en paralelo (uno en algoritmos, otro en lógica de negocio) sin generar conflictos de código, facilitando la escalabilidad del proyecto.