UNIVERSIDAD DE CALDAS

Faculty of Engineering

Technical Report:

Library Management System (LMS)

Prepared by:

Juan Camilo Cruz Parra

Jaime Andrés Cardona Díaz

Daner Alejandro Salazar Colorado

Submitted to:

Mario Alejandro Bravo Ortiz

Course:

Técnicas de Programación

Manizales, Caldas

2025

Contents

1 Library Management System (LMS) 2
1.1 General Description of the System . . . 2
1.2 Data Structures Used . . . 2
1.2.1 Lists and Dictionaries (Native) . . . 2
1.2.2 Stacks (LIFO - Last In, First Out) . . . 2
1.2.3 Waiting Structure (Reservation Management) . . . 3
1.3 Implemented Algorithms . . . 3
1.3.1 Sorting Algorithms . . . 3
1.3.2 Search Algorithms . . . 4
1.3.3 Recursion . . . 4
1.3.4 Backtracking (Combinatorial Optimization) . . . 5
1.4 Execution Results (Simulation) . . . 5
1.4.1 Case 1: Search and Loan . . . 5
1.4.2 Case 2: Bookshelf Optimization . . . 5
1.5 Justification of Technical Decisions . . . 6

1. Library Management System (LMS)

1.1 General Description of the System

The **Library Management System (LMS)** is a software application designed to manage the inventory flow, loans, and users of a library. The system is built using a **layered modular architecture**, clearly separating business logic, data structures, and utility algorithms.

The core of the system allows:

- Bulk loading of books from files (CSV/JSON).
- User management and loan control.
- Efficient search of bibliographic material.
- Generation of sorted reports.
- Solving optimization problems (bookshelf) and recursive statistical calculations.

The main interaction is done through a console interface (`Main.py`), which acts as the entry point and orchestrator of user requests to the main controller (`Biblioteca.py`).

1.2 Data Structures Used

The system implements both native and custom data structures to handle information efficiently.

1.2.1 Lists and Dictionaries (Native)

- **General Inventory (`List[Libro]`):** stores all books in the order in which they arrive. It allows $O(1)$ insertion complexity.
- **Sorted Inventory (`List[Libro]`):** parallel list maintained in ascending order by ISBN. It is essential for enabling binary search.
- **Users (`Dict[str, Usuario]`):** hash map that provides practically $O(1)$ access to users by their ID.

1.2.2 Stacks (LIFO - Last In, First Out)

A custom `Pila` class was implemented to manage each user's **loan history**.

Justification: a user's history is generally consulted to see "what they read last". A stack is the natural structure for this, allowing $O(1)$ access to the most recent loan without traversing the entire history.

```
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: Fragment from DataStructures/Pila.py

1.2.3 Waiting Structure (Reservation Management)

To handle out-of-stock books, the `PilaDeEspera` class is used. Although conceptually a queue (FIFO) would be the standard structure for turns, the system implements a waiting logic where requests are stacked.

Operation: when a book is returned, the system automatically checks this structure to assign the book to the next user in line according to the defined rules.

1.3 Implemented Algorithms

The project stands out for the manual implementation of classic computing algorithms, avoiding excessive use of "black box" functions from external libraries.

1.3.1 Sorting Algorithms

Merge Sort

Used to generate the **Inventory Report by Value**.

- **Complexity:** $O(n \log n)$ in all cases.
- **Justification:** chosen for its **stability** (it maintains the relative order of equal elements) and its guaranteed efficiency even with large volumes of data, unlike Quicksort which can degrade to $O(n^2)$.

```
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 by value

Insertion Sort

Used to sort the **loan history by date**.

- **Complexity:** $O(n^2)$ on average, but $O(n)$ on almost-sorted lists.
- **Justification:** loan histories are usually small and are built chronologically. Insertion Sort is extremely fast and memory-efficient for these almost-sorted data cases.

1.3.2 Search Algorithms

Binary Search

Used to search for books by **ISBN**.

- **Requirement:** the list must be sorted (it uses `inventario_ordenado_isbn`).
- **Complexity:** $O(\log n)$.
- **Impact:** allows finding a book among one million records in approximately 20 comparisons, versus one million comparisons with linear search.

```
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: Binary search by ISBN

Linear Search

Used to search by **title** or **author**.

- **Complexity:** $O(n)$.
- **Justification:** needed because we cannot keep the inventory sorted by all criteria simultaneously. It also allows partial matches (substrings).

1.3.3 Recursion

Stack Recursion

Implemented, for example, to find the **lightest book** by a given author. The function calls itself while processing the rest of the list and compares the result with the current element when "returning" from the recursion.

Tail Recursion

Implemented to compute the **average weight** of an author's books.

- **Logic:** it uses accumulators in the function parameters. The recursive call is the last instruction, which theoretically allows compilers/interpreters to optimize memory usage (*Tail Call Optimization*).

1.3.4 Backtracking (Combinatorial Optimization)

Used in the **bookshelf module** to select the best books under a weight constraint (Knapsack Problem).

- **Strategy:** depth-first exploration of the decision tree.
- **Pruning:** the algorithm immediately discards branches that cannot lead to a valid solution (for example, books weighing less than 0.5 kg or combinations that exceed the maximum accumulated weight), drastically reducing execution time.

```
if libro.peso < 0.5:
    continue  # PRUNING: Ignore very light books
if peso_actual + libro.peso <= max_peso:
    # Explore branch including the book...
    ...
```

Listing 1.4: Basic pruning in bookshelf module

1.4 Execution Results (Simulation)

Below are representative results based on system tests.

1.4.1 Case 1: Search and Loan

Input: user `"U001"` requests book with ISBN `"978-4"` (Modern AI).

Process:

1. `busqueda_binaria_isbn` locates `"978-4"` in very few steps.
2. The system checks stock. If `stock > 0`, it creates a `Prestamo` object and pushes it onto `U001.historial`.

Output: `Successful loan: 'Modern AI' delivered to Mario Bravo.`

1.4.2 Case 2: Bookshelf Optimization

Input: inventory with 5 books, maximum weight = 8.0 kg.

Process: the backtracking algorithm explores combinations:

- It discards *The Little Prince* (0.2 kg) due to the pruning rule (weight less than 0.5 kg).

- It evaluates the remaining combinations to maximize value in currency without exceeding 8 kg.

Output (example):

```
--- BEST SELECTION FOUND ---
Total Value: $420,000
Total Weight: 4.50 kg
Books:
 - Modern AI ($250,000, 2.5kg)
 - Clean Code ($180,000, 0.8kg)
 - Data Structures ($120,000, 1.2kg)
```

1.5 Justification of Technical Decisions

1. **Dual inventories:** it was decided to maintain two lists (`general` and `sorted`), sacrificing RAM to gain CPU speed. This allows fast insertions ($O(1)$ in the general list) and fast searches ($O(\log n)$ in the sorted list) at the same time.
2. **Use of a stack for history:** modeling the history as a stack is a user-centered design decision. In recommendation and query systems, recent activity has much more weight than older activity.
3. **Backtracking vs. brute force:** while both approaches can be compared, backtracking with pruning is the practical solution. Brute force (evaluating all combinations) is computationally infeasible ($O(2^n)$) as the inventory grows. Pruning significantly reduces this search space.
4. **Modularity:** separation into folders (`Controller`, `Entities`, `UtilityAlgorithms`) allows multiple developers to work in parallel (one on algorithms, another on business logic) without generating code conflicts, facilitating project scalability.