UNIVERSIDAD DE CALDAS

Faculty of Engineering

Analysis and Solution Strategies:

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. Analysis and Solution Strategies

In this document we explain the technical decisions and strategies we used to build the Library Management System (LMS). The idea is to justify why we chose certain data structures and algorithms to meet the project requirements.

1.1 Data Loading

The system needs to read information from different sources. Instead of forcing the use of a single file type, we created a function that automatically detects whether the file is CSV or JSON.

Implemented solution: The system looks at the file extension and decides how to read it. This is very useful because it allows us to work with Excel-style files (CSV) or with data coming from web applications (JSON) without having to manually transform anything.

```python
# We automatically detect the format
if ruta.endswith('.csv'):
    reader = csv.DictReader(f)  # For Excel-like files
elif ruta.endswith('.json'):
    data = json.load(f)         # For Web-style data
```

Listing 1.1: Automatic file format detection

1.2 Fast Searches: Double List

Searching for a book in an unsorted list is slow when there is a lot of data. To solve this, we used a two-list strategy.

Implemented solution:

- **General inventory:** stores the books in the order in which they arrived.
- **Sorted inventory:** stores the same books but sorted by ISBN.

Thanks to the sorted list, we can use **binary search**. Instead of checking book by book, the system splits the list and finds the book much faster.

```python
# Binary Search: much faster than checking one by one
while low <= high:
    mid = (low + high) // 2
    if inventario[mid].isbn == isbn:
        return mid  # Found!
    elif inventario[mid].isbn < isbn:
        low = mid + 1
    else:
        high = high - 1
```

Listing 1.2: Binary search on sorted inventory

1.3 Loan History (Stacks)

For the history of what a user has borrowed, we analyzed how that information is actually used. Generally, people want to see what they borrowed most recently.

Implemented solution: We used a **stack**. This structure works like a stack of plates: the last one placed is the first one taken. In this way, we always have the most recent loan at hand without having to search through the entire history.

```
# With the stack, the last item is always accessible
def apilar(self, item):
    self.items.append(item)

def desapilar(self):
    return self.items.pop()
```

Listing 1.3: Basic operations on the history stack

1.4 Reservations and Waiting List

When a book runs out, we need a way to organize the people who want it.

Implemented solution: We created a waiting list. The interesting part is that the system checks this list automatically: as soon as someone returns the book, the system immediately assigns it to the next person in line. This way, no time is wasted and the book keeps circulating efficiently.

1.5 Sorting Algorithms

We do not use the same method to sort everything; it depends on what is being sorted.

- **Merge Sort (for value reports):** used to sort all books by price. It is a very reliable algorithm that does not slow down even if the list is highly disordered.
- **Insertion Sort (for histories):** used to sort loans by date. Since histories grow gradually, they tend to be almost sorted. In this case, this algorithm is faster and simpler than more complex alternatives.

1.6 Optimizing the Bookshelf

We had the challenge of choosing the best books for a shelf that cannot hold much weight.

Implemented solution (backtracking): The system tries different combinations of books to see which one yields the highest value without exceeding the weight limit. To avoid spending too much time on low-value combinations, we added a pruning rule: if a book is very light (less than 0.5 kg), it is skipped immediately. This makes the calculation much faster.

```python
# If the book is very light, we skip it to save time
if libro.peso < 0.5:
    continue

if peso_actual + libro.peso <= max_peso:
    # We test whether this combination works
    ...
```

Listing 1.4: Pruning of low-relevance combinations

1.7 Recursion

We use recursive functions (functions that call themselves) for some calculations.

- **Stack recursion:** used to find the lightest book. It is a clean way to traverse the list step by step.
- **Tail recursion:** used to compute the average weight. It is an efficient technique that helps the program avoid using too much memory while performing the sums, since the recursive call is the last operation.

1.8 Conclusion

Every technical decision has a reason behind it. We chose these data structures and algorithms so that the system would be fast, easy to use, and capable of handling different situations without issues. The combination of a double list for fast searches, stacks for histories, automated waiting lists, context-aware sorting algorithms, and backtracking optimized with pruning enables the Library Management System (LMS) to meet the functional requirements with a solid and maintainable design.