## Exercise 6: Library Management System

### Step 1: Understand Search Algorithms

### Explain linear search and binary search algorithms.

#### Linear Search

* **Description**: Linear search is a simple search algorithm that checks every element in the list sequentially until the target element is found or the list ends.
* **Time Complexity**:
  + Best Case: O(1) (if the target is the first element)
  + Average/Worst Case: O(n) (if the target is in the middle or not present)
* **Use Case**: Effective for small, unsorted datasets.

#### Binary Search

* **Description**: Binary search is an efficient algorithm for finding an item from a sorted list. It works by repeatedly dividing the search interval in half. If the target value is less than the middle element, the search continues in the left half; otherwise, it continues in the right half.
* **Time Complexity**:
  + Best/Average/Worst Case: O(log n)
* **Use Case**: Effective for large, sorted datasets.

### Step 2: Setup

#### Define the Book Class

// Java implementation

public class Book {

private String bookId;

private String title;

private String author;

public Book(String bookId, String title, String author) {

this.bookId = bookId;

this.title = title;

this.author = author;

}

// Getters

public String getBookId() {

return bookId;

}

public String getTitle() {

return title;

}

public String getAuthor() {

return author;

}

}

### Step 3: Implementation

#### Linear Search Implementation

// Java implementation of Linear Search

import java.util.ArrayList;

import java.util.List;

public class Library {

private List<Book> books;

public Library() {

this.books = new ArrayList<>();

}

public void addBook(Book book) {

books.add(book);

}

public Book linearSearchByTitle(String title) {

for (Book book : books) {

if (book.getTitle().equalsIgnoreCase(title)) {

return book;

}

}

return null;

}

}

#### Binary Search Implementation

// Java implementation of Binary Search

import java.util.Collections;

import java.util.Comparator;

public class Library {

private List<Book> books;

public Library() {

this.books = new ArrayList<>();

}

public void addBook(Book book) {

books.add(book);

}

// Sort the books by title before performing binary search

public void sortBooksByTitle() {

Collections.sort(books, new Comparator<Book>() {

public int compare(Book b1, Book b2) {

return b1.getTitle().compareToIgnoreCase(b2.getTitle());

}

});

}

public Book binarySearchByTitle(String title) {

int left = 0;

int right = books.size() - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

Book midBook = books.get(mid);

int comparison = midBook.getTitle().compareToIgnoreCase(title);

if (comparison == 0) {

return midBook;

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

### Step 4: Analysis

#### Time Complexity Comparison

* **Linear Search**:
  + Best Case: O(1) (if the target is the first element)
  + Average/Worst Case: O(n) (if the target is in the middle or not present)
* **Binary Search**:
  + Best/Average/Worst Case: O(log n)

#### When to Use Each Algorithm

* **Linear Search**:
  + Use for small datasets where the overhead of sorting is not justified.
  + Useful when the dataset is unsorted and small enough that performance is not significantly impacted by O(n) complexity.
* **Binary Search**:
  + Use for large, sorted datasets where the logarithmic complexity provides significant performance benefits.

Requires an initial sorting step (O(n log n)), but subsequent searches are efficient.