**Exercise 6: Library Management System**

**Scenario:**

You are developing a library management system where users can search for books by title or author.

**Understanding Search Algorithms**

**1. Linear Search**

* **Definition**: Linear search scans each element in the list one by one until the target is found or the end is reached.
* **Use Case**: Best suited for **unsorted** data.
* **Time Complexity**:
  + Best Case: O(1) (target found at first element)
  + Worst Case: O(n)
  + Average Case: O(n)

**2. Binary Search**

* **Definition**: Binary search divides the sorted array into halves and repeatedly compares the middle element with the target.
* **Precondition**: Data **must be sorted**.
* **Time Complexity**:
  + Best Case: O(1)
  + Worst Case: O(log n)
  + Average Case: O(log n)

**Setup :-**

**Booj.java :-**

public class Book {

private int bookId;

private String title;

private String author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

public int getBookId() { return bookId; }

public String getTitle() { return title; }

public String getAuthor() { return author; }

@Override

public String toString() {

return bookId + ": " + title + " by " + author;

}

}

**Implementation :-**

**LinearSearch.java :-**

public class LinearSearch {

public static Book search(Book[] books, String title) {

for (Book book : books) {

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

return book;

}

}

return null;

}

}

**BinarySearch.java :-**

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

public static Book search(Book[] books, String title) {

Arrays.sort(books, Comparator.comparing(Book::getTitle)); // Sort first

int left = 0, right = books.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

int cmp = books[mid].getTitle().compareToIgnoreCase(title);

if (cmp == 0) return books[mid];

else if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

}

**Main.java :-**

public class Main {

public static void main(String[] args) {

Book[] library = {

new Book(101, "The Alchemist", "Paulo Coelho"),

new Book(102, "Atomic Habits", "James Clear"),

new Book(103, "Clean Code", "Robert C. Martin"),

new Book(104, "Deep Work", "Cal Newport")

};

System.out.println("Linear Search:");

Book found1 = LinearSearch.search(library, "Clean Code");

System.out.println(found1 != null ? found1 : "Book not found");

System.out.println("\nBinary Search:");

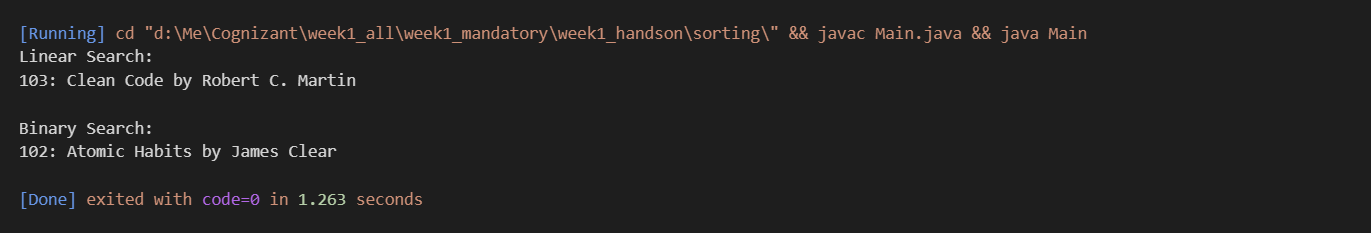
Book found2 = BinarySearch.search(library, "Atomic Habits");

System.out.println(found2 != null ? found2 : "Book not found");

}

}

**Output :-**



**Analysis: Linear Search vs Binary Search**

**Time Complexity Comparison**

| **Operation Type** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Best Case** | O(1) | O(1) |
| **Average Case** | O(n) | O(log n) |
| **Worst Case** | O(n) | O(log n) |
| **Space Complexity** | O(1) | O(1) |

**Explanation:**

**🔹 Linear Search**

* **Best Case (O(1))**: The element is found at the first position.
* **Average Case (O(n/2)) → O(n)**: On average, the element is found halfway through the list.
* **Worst Case (O(n))**: The element is at the end or not present at all, requiring full traversal.
* **No requirement for sorting**: Works on both **sorted** and **unsorted** arrays.
* **Simple to implement**: Ideal for quick prototypes or small data.

**🔹 Binary Search**

* **Best Case (O(1))**: The element is at the middle index during the first comparison.
* **Average/Worst Case (O(log n))**: The search space is divided by two in every iteration.
* **Requires Sorted Data**: Operates only on **sorted arrays or collections**.
* **Efficient for large datasets**: Performance scales much better than linear search with increased data size.

**When to Use Each Algorithm**

**🔹 Use Linear Search When:**

1. **Data is Unsorted**:
   * If your dataset is dynamic or frequently changing, it might not be feasible to sort before every search.
2. **Small Dataset**:
   * For a small number of items (less than 100–1000 elements), the simplicity of linear search often outweighs the performance gains from binary search.
3. **Searching by Multiple Fields**:
   * If you’re searching by complex conditions (e.g., part of title, multiple attributes), linear search allows more flexibility.

**Example**:

// Searching for "Deep Work" in unsorted book list

Book book = LinearSearch.search(books, "Deep Work");

**🔹 Use Binary Search When:**

1. **Data is Sorted**:
   * The dataset must be sorted (either ascending or descending) based on the search key.
2. **Large Dataset**:
   * For large arrays (thousands or millions of items), binary search drastically reduces search time.
3. **Frequent Read Operations**:
   * In read-heavy applications like search engines, indexing systems, or libraries where searches are common but data changes infrequently.