**Module 1: ALGORITHMS\_DATA STRUCTURES**

**Exercise 2: E-commerce Platform Search Function**

**Step 1:** Understand Asymptotic Notation

* **Big O Notation** measures how algorithm runtime or space grows with input size.
* **Sequential (Linear) Search:** O(n) — Checks each item until found.
* **Binary Search:** O(log n) — Requires sorted data; divides search space in half each step.

**Step 2:** Setup – Item Class

class Item {

int itemCode;

String itemName;

String itemCategory;

Item(int itemCode, String itemName, String itemCategory) {

this.itemCode = itemCode;

this.itemName = itemName;

this.itemCategory = itemCategory;

}

}

**Step 3:** Implementation – Sequential & Binary Search

import java.util.Arrays;

public class SearchDemo {

public static void main(String[] args) {

Item[] inventory = {

new Item(204, "Keyboard", "Electronics"),

new Item(201, "Smartphone", "Electronics"),

new Item(202, "T-Shirt", "Apparel"),

new Item(203, "Notebook", "Stationery")

};

Arrays.sort(inventory, (a, b) -> Integer.compare(a.itemCode, b.itemCode));

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

Item resultSeq = sequentialSearch(inventory, 203);

showItem(resultSeq);

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

Item resultBin = binarySearch(inventory, 203);

showItem(resultBin);

}

public static Item sequentialSearch(Item[] inventory, int targetCode) {

for (Item item : inventory) {

if (item.itemCode == targetCode) {

return item;

}

}

return null;

}

public static Item binarySearch(Item[] inventory, int targetCode) {

int start = 0, end = inventory.length - 1;

while (start <= end) {

int mid = start + (end - start) / 2;

if (inventory[mid].itemCode == targetCode) {

return inventory[mid];

} else if (inventory[mid].itemCode < targetCode) {

start = mid + 1;

} else {

end = mid - 1;

}

}

return null;

}

public static void showItem(Item item) {

if (item != null) {

System.out.println("Item Found:");

System.out.println("Code: " + item.itemCode + ", Name: " + item.itemName + ", Category: " + item.itemCategory);

} else {

System.out.println("Item not found.");

}

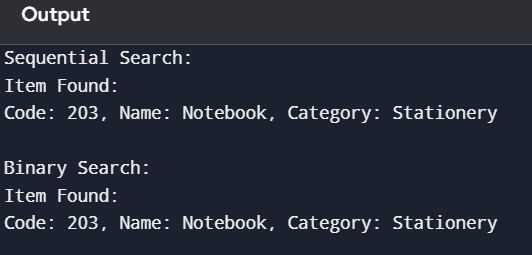
}

}

**Step 4:** Analysis – Time Complexity

* **Sequential Search:** O(n) — Checks each element until match or end.
* **Binary Search:** O(log n) — Efficient search but requires sorted array.

**Output:**

****

**Exercise 7: Financial Forecasting**

**Step 1:** Recursive Algorithms

Recursion is a method where a function calls itself to solve smaller sub-problems of a larger task.

In financial forecasting, we often want to calculate how a value (such as an investment or savings) grows over time. If we reinvest interest every year, the growth is compound, and can be expressed recursively.

**Step 2 & 3:** Recursive Implementation – Future Value Prediction

Formula:

Future Value = Present Value × (1 + rate)^n

**Code:**

public class ForecastApp {

public static void main(String[] args) {

double currentAmount = 1500.0;

double annualRate = 0.08; // 8% annual growth

int timePeriod = 6;

double projectedAmount = calculateFutureValue(currentAmount, annualRate, timePeriod);

System.out.printf("Future Value after %d years = ₹%.2f\n", timePeriod, projectedAmount);

}

public static double calculateFutureValue(double amount, double rate, int years) {

if (years == 0) {

return amount;

}

return calculateFutureValue(amount \* (1 + rate), rate, years - 1);

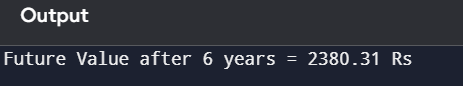
}

}

**Step 4:** Analysis – Time Complexity

* Time Complexity: O(n) – one recursive call per year
* Space Complexity: O(n) – call stack stores n recursive calls.

**Output:**

****