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

**What is Big O Notation?**

* Big O notation describes the upper bound of an algorithm's runtime or space complexity as the input size grows.
* It helps you evaluate scalability and performance of algorithms.

**Big O in Search:**

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

* **Best Case:** When the item is found early.
* **Average Case:** Random distribution of input.
* **Worst Case:** When the item is at the end (or not found).

**Step 2: Setup a Product Class:**

**Product.java:**

package ECommerceSearch;

public class Product {

int productId;

String productName;

String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

*@Override*

public String toString() {

return "ProductID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

**Step 3: Implement Linear and Binary Search:**

**Main.java:**

package ECommerceSearch;

import java.util.Arrays;

import java.util.Comparator;

public class Main {

public static Product linearSearch(Product[] products, String name) {

for (Product p : products) {

if (p.productName.equalsIgnoreCase(name)) {

return p;

}

}

return null;

}

public static Product binarySearch(Product[] products, String name) {

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

while (left <= right) {

int mid = (left + right) / 2;

int compare = products[mid].productName.compareToIgnoreCase(name);

if (compare == 0) return products[mid];

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

else right = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(201, "Laptop", "Electronics"),

new Product(202, "Shoes", "Footwear"),

new Product(203, "Camera", "Electronics"),

new Product(204, "Watch", "Accessories"),

new Product(205, "Backpack", "Travel"),

new Product(206, "Sunglasses", "Accessories")

};

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

System.out.println(linearSearch(products, "Watch"));

System.out.println(linearSearch(products, "Sunglasses"));

System.out.println(linearSearch(products, "Tablet")); // Not found

Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

System.out.println("\n Binary Search (Sorted):");

System.out.println(binarySearch(products, "Camera"));

System.out.println(binarySearch(products, "Laptop"));

System.out.println(binarySearch(products, "Mobile")); // Not found

}

}

**Step 4: Analysis:**

**Time Complexity:**

| **Algorithm** | **Time Complexity** | **Sorted Required?** |
| --- | --- | --- |
| **Linear Search** | O(n) | No |
| **Binary Search** | O(log n) | Yes |

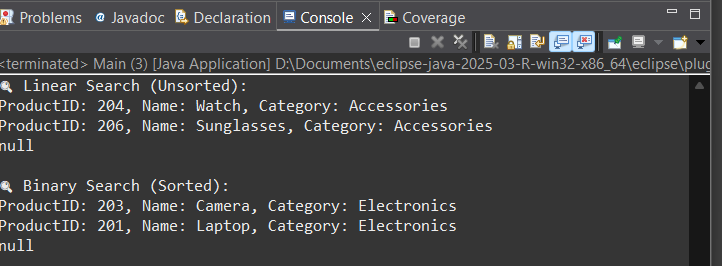
* Linear Search is simple, works on any list.
* Binary Search is much faster but needs the data to be sorted.

**Which is Better for an E-commerce Platform?**

Binary Search is better if data is sorted or indexed (e.g., via database or search engine).For small datasets or unsorted arrays, linear search might be acceptable.

In practice, e-commerce platforms use indexes and search engines (like Elasticsearch), but conceptually binary search fits better for optimized search logic.

**Output:**

****

**Exercise 7: Financial Forecasting:**

**Step 1: Understand Recursive Algorithms**

**What is Recursion?**

* Recursion is a programming technique where a method calls itself to solve a smaller instance of a problem.
* It’s useful when a problem can be broken into smaller subproblems of the same type.

**Why use recursion for forecasting?**

* Forecasting involves applying a growth rate repeatedly over a number of periods (e.g., years).
* This repetitive pattern fits naturally with recursion.

**Step 2: Setup — Recursive Future Value Method**

Let’s forecast future value with the formula:

**FV=P×(1+r)**

Where:

* P = present value
* r = growth rate (in decimal, e.g., 5% = 0.05)
* n = number of years

**Main.java:**

public class Main {

public static double futureValue(double presentValue, double rate, int years) {

if (years == 0) {

return presentValue;

}

return futureValue(presentValue \* (1 + rate), rate, years - 1);

}

public static void main(String[] args) {

double initialAmount = 10000;

double growthRate = 0.07;

int years = 5;

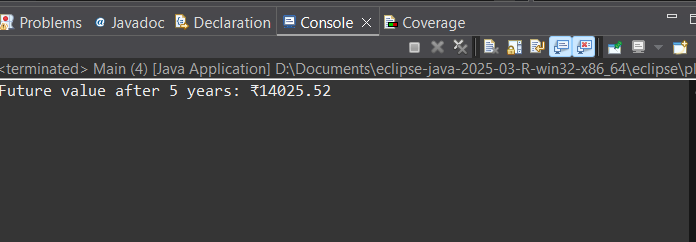
double result = futureValue(initialAmount, growthRate, years);

System.out.printf("Future value after %d years: ₹%.2f\n", years, result);

}

}

**Output:**

****

**Step 4: Analysis:**

**Time Complexity:**

* The recursive method calls itself once per year.
* **So time complexity is:** T(n)=O(n)T(n) = O(n)T(n)=O(n)

where n = number of years.

**Optimized Iterative Version:**

public static double futureValueIterative(double presentValue, double rate, int years) {

for (int i = 0; i < years; i++) {

presentValue \*= (1 + rate);

}

return presentValue;

}

**Key Differences:**

| **Feature** | **Recursive Version** | **Iterative Version** |
| --- | --- | --- |
| **Simplicity** | Easy to read | Easy to optimize |
| **Time Complexity** | O(n) | O(n) |
| **Space Complexity** | O(n) (stack frames) | O(1) |
| **Optimization Need** | Not urgent | More efficient for large n |