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

**Big O Notation:**

Big O notation describes how the runtime of an algorithm increases relative to the size of the input (n). It helps us analyze and compare algorithms, especially for large inputs.

Examples:

* O(1) → Constant time
* O(log n) → Logarithmic time
* O(n) → Linear time
* O(n log n) → Log-linear time

**Best, Average, and Worst-Case Scenarios for Search:**

| **Case** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Best** | O(1) (found at start) | O(1) (middle element) |
| **Average** | O(n/2) ≈ O(n) | O(log n) |
| **Worst** | O(n) | O(log n) |

import java.util.Arrays;

import java.util.Comparator;

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;

}

//toString method to display product details

public String toString() {

return productId + " - " + productName + " (" + category + ")";

}

}

class SearchOperations {

// Linear Search

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

for (int i = 0; i < products.length; i++) {

if (products[i].productName.equalsIgnoreCase(productName)) {

return i;

}

}

return -1;

}

// Binary Search

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

// Sort products by productName

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

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

while (left <= right) {

int mid = (left + right) / 2;

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

if (result == 0)

return mid;

else if (result > 0)

left = mid + 1;

else

right = mid - 1;

}

return -1;

}

}

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(1, "Phone", "Electronics"),

new Product(2, "Shirt", "Clothing"),

new Product(3, "Laptop", "Electronics"),

new Product(4, "Shoes", "Footwear"),

new Product(5, "Watch", "Accessories")

};

// Product to search

String searchProduct = "Laptop";

// Linear Search

int linearIndex = SearchOperations.linearSearch(products, searchProduct);

if (linearIndex != -1)

System.out.println("Linear Search: Found at index " + linearIndex + " → " + products[linearIndex]);

else

System.out.println("Linear Search: Not Found");

// Binary Search

int binaryIndex = SearchOperations.binarySearch(products, searchProduct);

if (binaryIndex != -1)

System.out.println("Binary Search: Found at index " + binaryIndex + " → " + products[binaryIndex]);

else

System.out.println("Binary Search: Not Found");

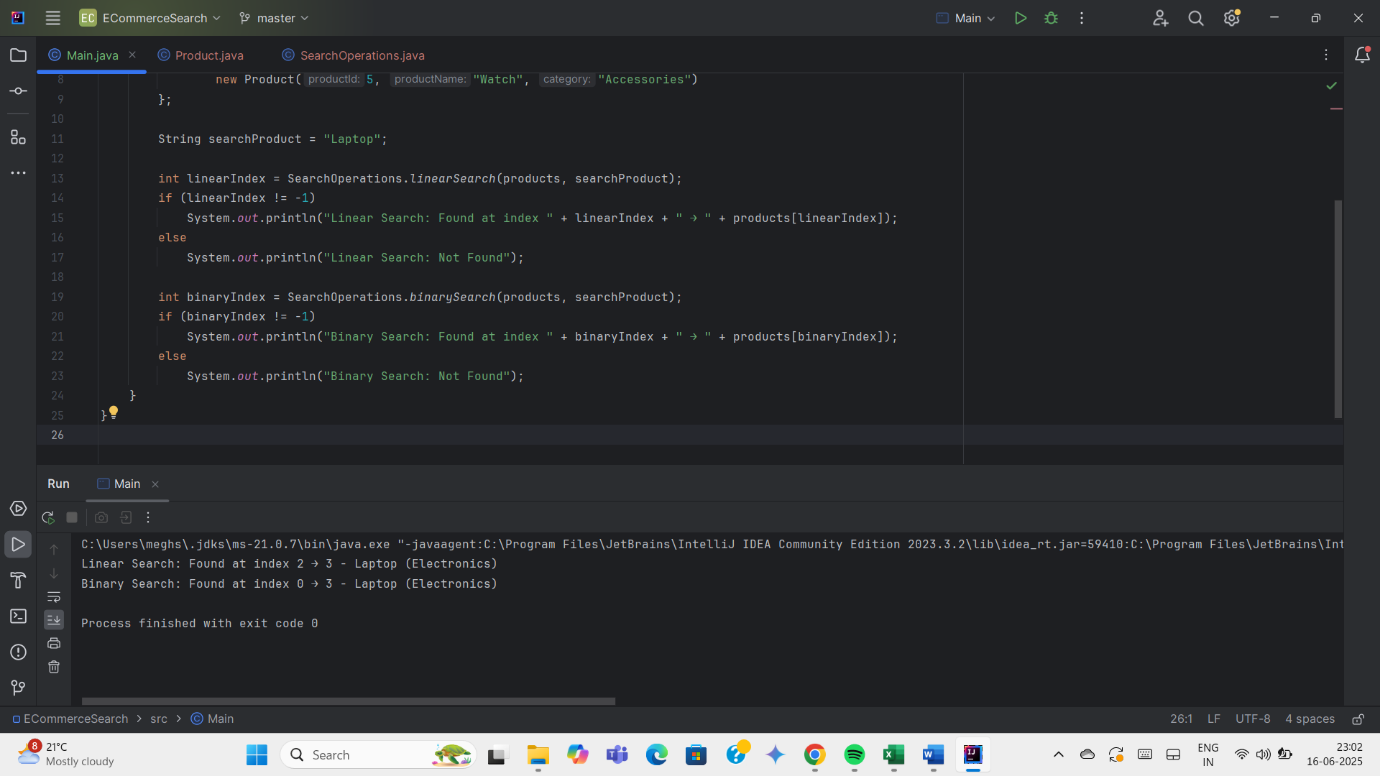
}

}

**Analysis:**

| **Algorithm** | **Time Complexity** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- | --- |
| Linear Search | O(n) | O(1) | O(n/2) ≈ O(n) | O(n) |
| Binary Search | O(log n) | O(1) | O(log n) | O(log n) |

* **Binary Search** is much faster on **sorted data** due to O(log n) performance.
* **Linear Search** works on **unsorted data** but is slower for large n.



**Exercise 7: Financial Forecasting**

Recursion is a technique where a method calls itself to solve a smaller instance of the same problem until a base condition is met.

Example Problem: Calculating factorial, Fibonacci sequence

* Simplifies code for problems with repetitive sub-structures.
* Clean and elegant solutions for divide-and-conquer problems.

public class ForecastTool {

public static double forecastValue(double currentValue, double growthRate, int years) {

// Base condition

if (years == 0) {

return currentValue;

}

//increase value by growthRate and decrease years

return forecastValue(currentValue \* (1 + growthRate), growthRate, years - 1);

}

public static void main(String[] args) {

double initialValue = 10000; // Initial amount

double growthRate = 0.08; // 8% annual growth

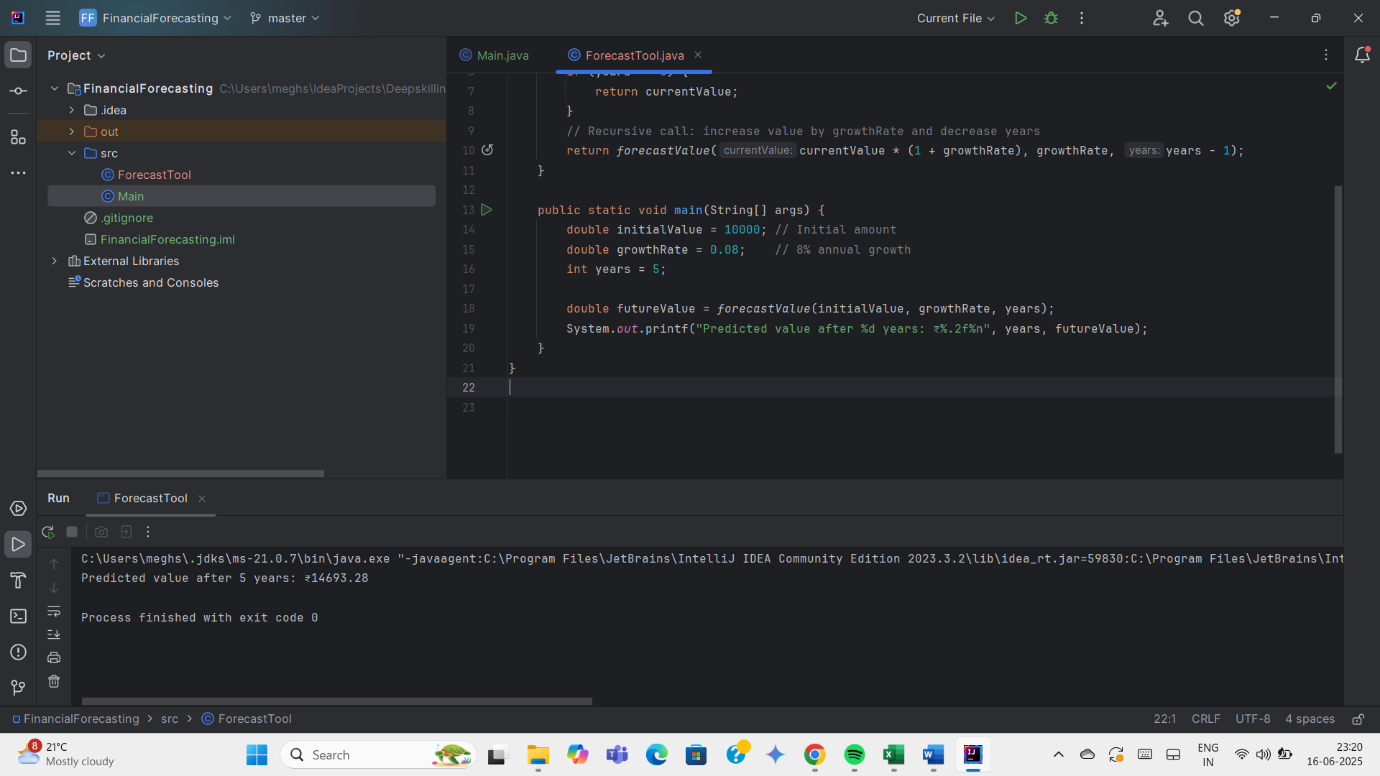
int years = 5;

double futureValue = forecastValue(initialValue, growthRate, years);

System.out.printf("Predicted value after %d years: ₹%.2f%n", years, futureValue);

}

}

****

**Analysis:**

| **Aspect** | **Value** |
| --- | --- |
| **Time Complexity** | O(n) — one recursive call per year |
| **Space Complexity** | O(n) — due to function call stack |

Recursive calls use stack space and can cause a StackOverflowError for large n.

**Optimization Techniques:**

**Tail Recursion** (if language/JVM optimizes it)

**Memoization** - Store results of subproblems so you don’t compute the same result multiple times.It helpswhen the recursion has overlapping subproblems like Fibonacci or DP problems.

**Iterative Solution** - If recursion is linear (like in financial forecasting or factorial), an iterative loop is much faster and memory-efficient since it avoids the call stack overhead.

**Closed-Form Formula** - A closed-form formula is a direct mathematical expression that gives the result in a single calculation, without the need for recursion or looping. It is most efficient in terms of speed and scalability.