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

**Understand Asymptotic Notation:**

Big O Notation is used to describe the performance or complexity of an algorithm. It tells us how the runtime or space requirements grow with the size of the input. It focuses on the upper bound of the running time, helping developers anticipate worst-case performance.

|  |  |  |  |
| --- | --- | --- | --- |
| Search Type | Best Case | Average Case | Worst Case |
| Linear Search | Element is at index 0 → O(1) | Element is in middle → O(n/2) ≈ O(n) | Element not found or at last → O(n) |
| Binary Search | Middle element is match → O(1) | Mid keeps halving → O(log n) | Element not found → O(log n) |

**Setup:**  
**Code:**

public class Product {

private int productId;

private String productName;

private String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public String getProductName() {

return productName;

}

*@Override*

public String toString() {

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

}

}

**Implementation:  
  
Code:**

import java.util.Arrays;

import java.util.Comparator;

import java.util.Scanner;

public class ECommerceSearch {

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

for (Product product : products) {

if (product.getProductName().equalsIgnoreCase(targetName)) {

return product;

}

}

return null;

}

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

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

String midName = products[mid].getProductName().toLowerCase();

int cmp = midName.compareTo(targetName.toLowerCase());

if (cmp == 0) {

return products[mid];

} else if (cmp < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(101, "Shoes", "Footwear"),

new Product(102, "T-shirt", "Apparel"),

new Product(103, "Laptop", "Electronics"),

new Product(104, "Book", "Stationery"),

new Product(105, "Mobile", "Electronics")

};

Scanner scanner = new Scanner(System.***in***);

System.***out***.print("Enter product name to search: ");

String userInput = scanner.nextLine();

// Linear Search

System.***out***.println("\nLinear Search");

Product resultLinear = *linearSearch*(products, userInput);

if (resultLinear != null) {

System.***out***.println("Product found: " + resultLinear);

} else {

System.***out***.println("Product not found.");

}

// Binary Search (requires sorted array)

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

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

Product resultBinary = *binarySearch*(products, userInput);

if (resultBinary != null) {

System.***out***.println("Product found: " + resultBinary);

} else {

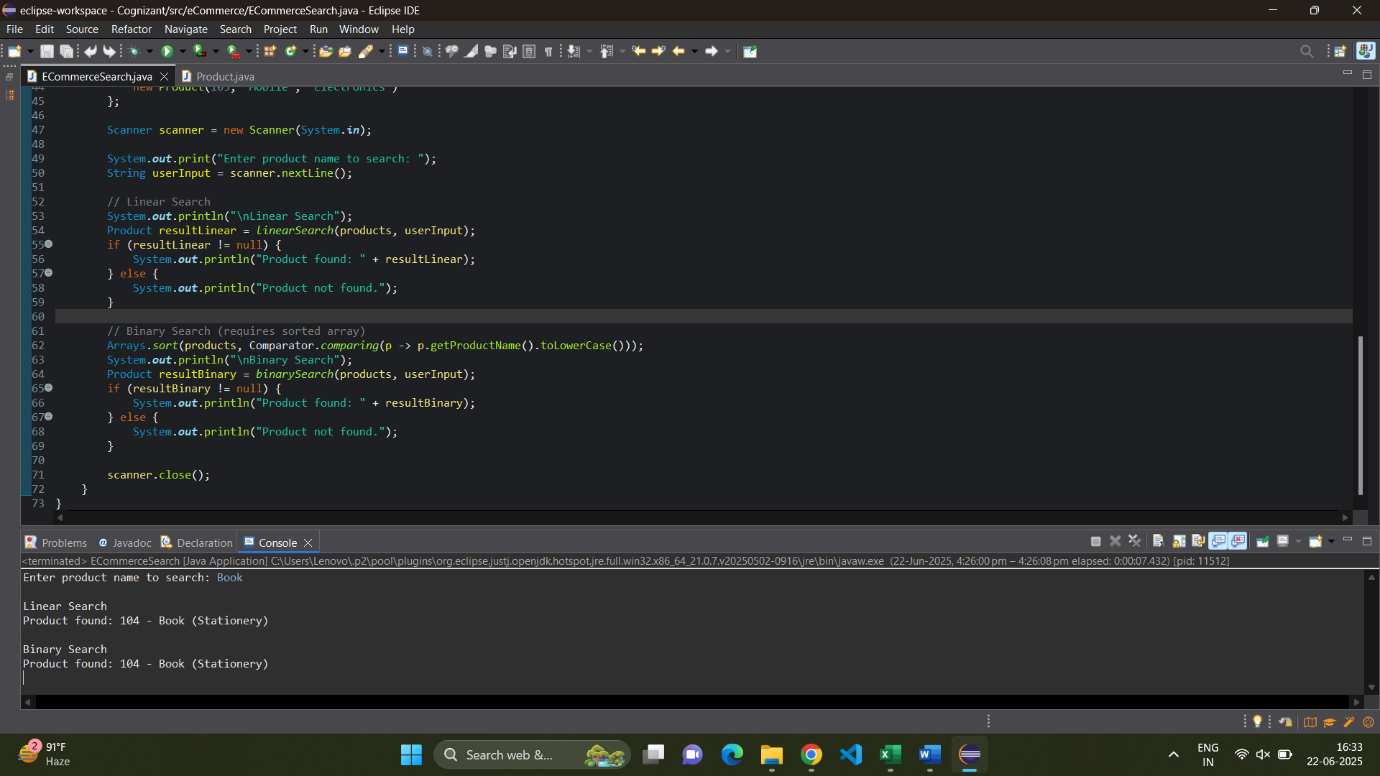
System.***out***.println("Product not found.");

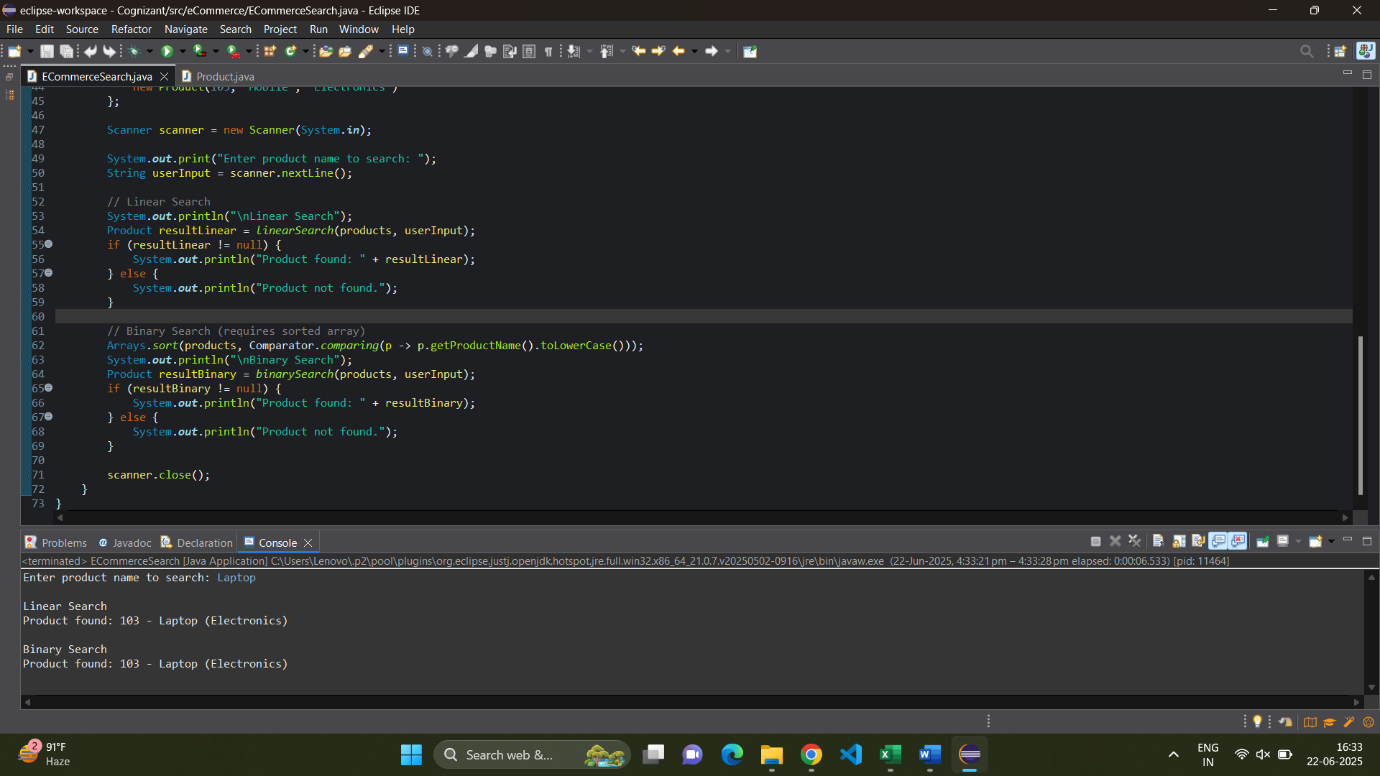
}

scanner.close();

}

}

**Output:  
  
**

****

**Analysis:**

1. Linear Search:

* Best Case: O(1) → First item is the target
* Worst Case: O(n) → Last item or not found
* Average Case: O(n)

2. Binary Search:

* Best Case: O(1) → Middle element is the target
* Worst Case: O(log n)
* Average Case: O(log n)
* Binary Search is better for large sorted datasets.
* Linear Search is simple and works on unsorted data.
* For an e-commerce platform, Binary Search (or even advanced data structures like HashMaps or Trees) is more scalable.

**Exercise 7: Financial Forecasting  
  
Understanding Recursive Algorithms**

**Recursion** is a programming technique where a function **calls itself** to solve smaller instances of the same problem until it reaches a base case.

It helps in **breaking down complex problems** like tree traversal, factorial, Fibonacci, etc. For financial forecasting, it can simplify predictions by applying **repetitive growth formulas** over years.

**Setup:**

We'll calculate future value using the formula:

FV=PV×(1+r)n

where:

FV = Future Value

PV = Present Value

r = growth rate (e.g., 5% as 0.05)

n = number of years

We’ll apply recursion to implement this.

**Implementation:**

**Code:**

import java.util.Scanner;

public class FinancialForecast {

// Recursive method to calculate future value

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

if (years == 0) {

return presentValue;

}

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

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.***in***);

System.***out***.print("Enter Present Value (₹): ");

double presentValue = scanner.nextDouble();

System.***out***.print("Enter Annual Growth Rate (in %, e.g., 5 for 5%): ");

double ratePercent = scanner.nextDouble();

System.***out***.print("Enter Number of Years: ");

int years = scanner.nextInt();

// Convert percentage to decimal

double growthRate = ratePercent / 100;

// Calculate Future Value using recursion

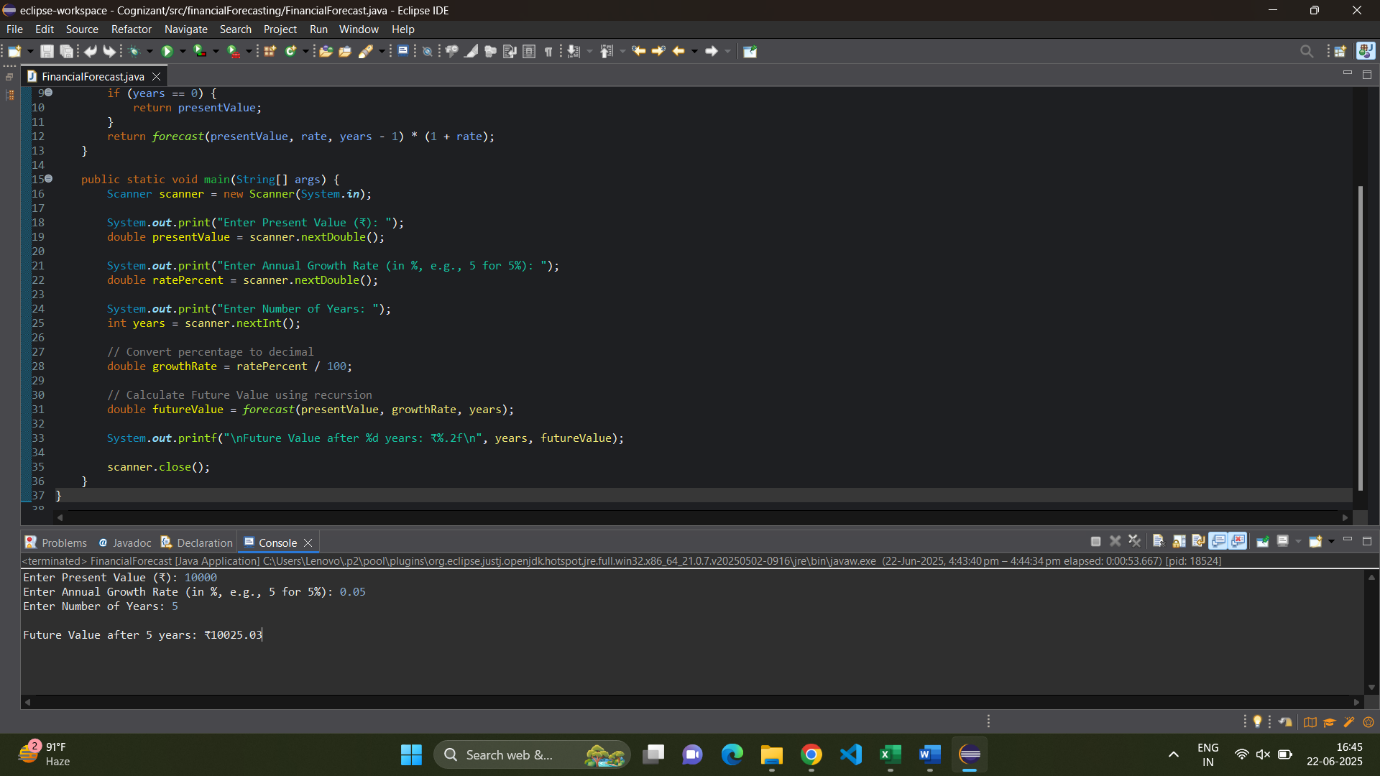
double futureValue = *forecast*(presentValue, growthRate, years);

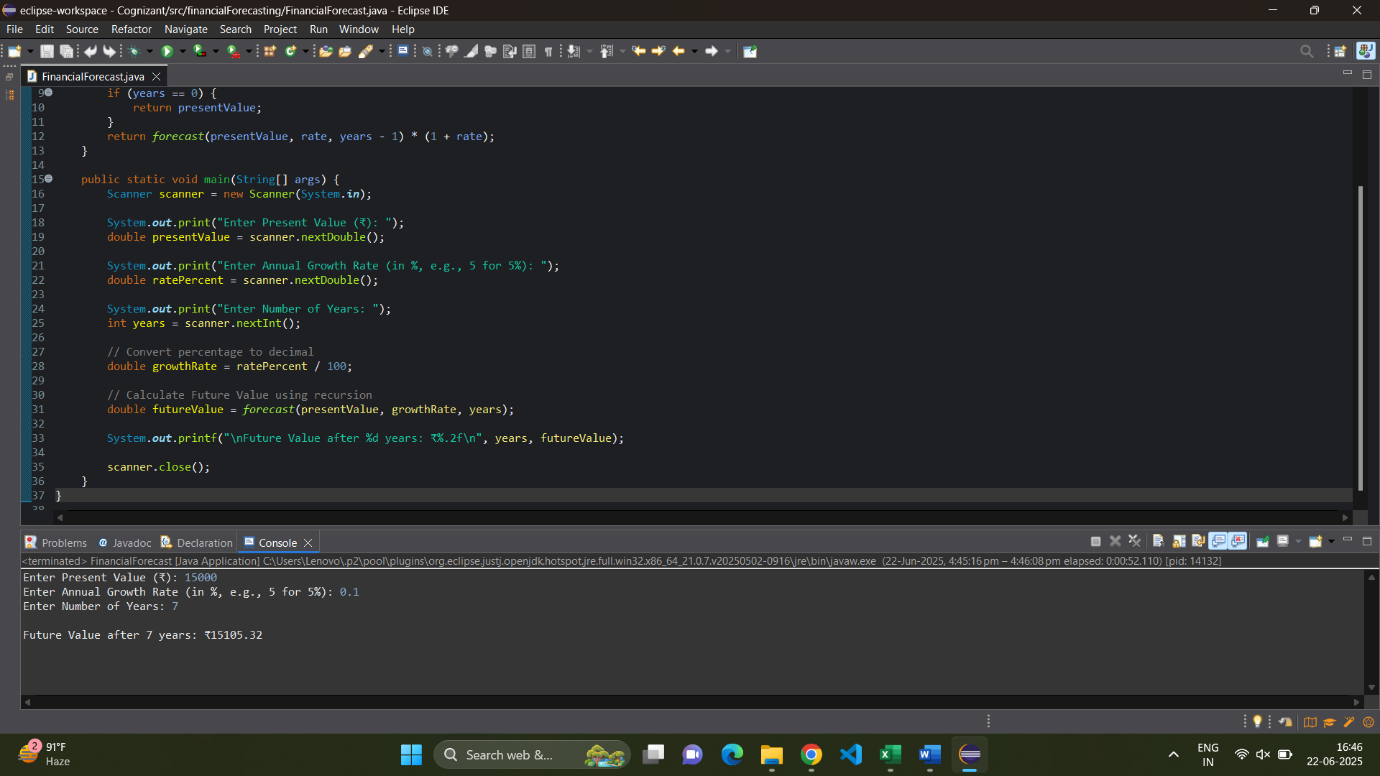
System.***out***.printf("\nFuture Value after %d years: ₹%.2f\n", years, futureValue);

scanner.close();

}

}

**Output:  
  
**

****

**Analysis:**

**Time Complexity**

**Recursive Depth** = n (number of years)

Only one recursive call per level → **O(n)** time

No overlapping subproblems, so **no redundant computations.**

**How to Optimize?**

* Option 1: Memoization (not needed here as no overlapping subproblems)
* Option 2: Use Iteration for Performance

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

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

presentValue \*= (1 + rate);

}

return presentValue;

}