Algorithms\_Data Structures

**Mandatory HandsOn**

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

* Big O notation describes the **time complexity** of an algorithm in terms of **input size (n)**.
* It tells how the performance of an algorithm changes as the input grows.

|  |  |  |  |
| --- | --- | --- | --- |
| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| Linear Search | O(1) | O(n/2) =O(n) | O(n) |
| Binary Search | O(1) | O(Log n) | O(Log n) |

**Product.java**

package com.ecommerce.search;

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;

}

}

**SearchOperations.java**

package com.ecommerce.search;

public class SearchOperations {

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

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

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

return i; // return index

}

}

return -1; // not found

}

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

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

while (left <= right) {

int mid = (left + right) / 2;

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

if (comparison == 0) return mid;

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

else right = mid - 1;

}

return -1;

}

}

**Main.java**

package com.ecommerce.search;

import java.util.Arrays;

import java.util.Comparator;

public class Main {

public static void main(String[] args) {

Product[] products = {

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

new Product(2, "Shoes", "Fashion"),

new Product(3, "Book", "Education"),

new Product(4, "Phone", "Electronics")

};

// Linear Search

int linearIndex = SearchOperations.*linearSearch*(products, "Phone");

if (linearIndex != -1) {

System.*out*.println("Linear Search: Phone found in position " + (linearIndex + 1));

} else {

System.*out*.println("Linear Search: Phone not found.");

}

// Sort products by name for Binary Search

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

// Binary Search

int binaryIndex = SearchOperations.*binarySearch*(products, "Phone");

if (binaryIndex != -1) {

System.*out*.println("Binary Search: Phone found in sorted position " + (binaryIndex + 1));

} else {

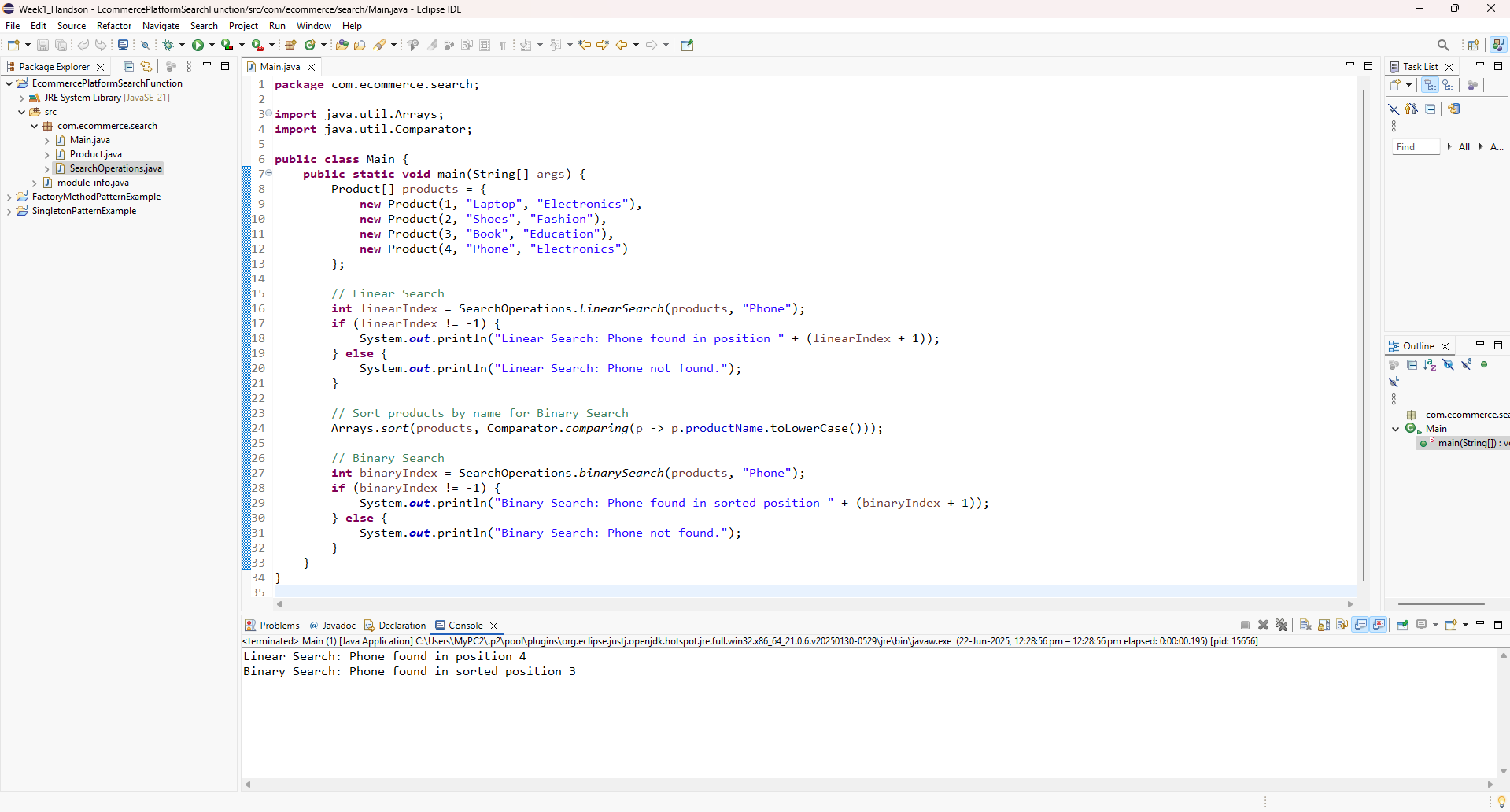
System.*out*.println("Binary Search: Phone not found.");

}

}

}

**Output**



**Time Complexity**

| **Algorithm** | **Time Complexity (Best / Avg / Worst)** |
| --- | --- |
| Linear Search | O(1) / O(n) / O(n) |
| Binary Search | O(1) / O(log n) / O(log n) |

**Suitability for E-commerce**

* **Use Binary Search**:
  + When product data is **sorted** (e.g., alphabetically by name or ID).
  + Much **faster** for large datasets.
* **Use Linear Search**:
  + When data is **unsorted** or **small in size**.
  + Simpler but less efficient for big datasets.

**Exercise 7: Financial Forecasting**

**Recursion** is a programming technique where a function calls itself to solve smaller instances of the same problem.

* **Base Case:** The condition under which recursion stops.
* **Recursive Case:** The function calls itself with a modified argument that moves towards the base case.

**FinancialForecastApp.java**

package com.financial.forecasting;

public class FinancialForecastApp {

public static void main(String[] args) {

double presentValue = 10000;

double growthRate = 0.05;

int years = 5;

ForecastCalculator calculator = new ForecastCalculator();

double forecastRecursive = calculator.futureValueRecursive(presentValue, growthRate, years);

double forecastIterative = calculator.futureValueIterative(presentValue, growthRate, years);

System.*out*.println("Future Value (Recursive): " + forecastRecursive);

System.*out*.println("Future Value (Iterative): " + forecastIterative);

}}

**ForecastCalculator.java**

package com.financial.forecasting;

public class ForecastCalculator {

// Recursive method

public double futureValueRecursive(double pv, double rate, int years) {

if (years == 0) {

return pv;

}

return futureValueRecursive(pv \* (1 + rate), rate, years - 1);

}

// Iterative method

public double futureValueIterative(double pv, double rate, int years) {

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

pv \*= (1 + rate);

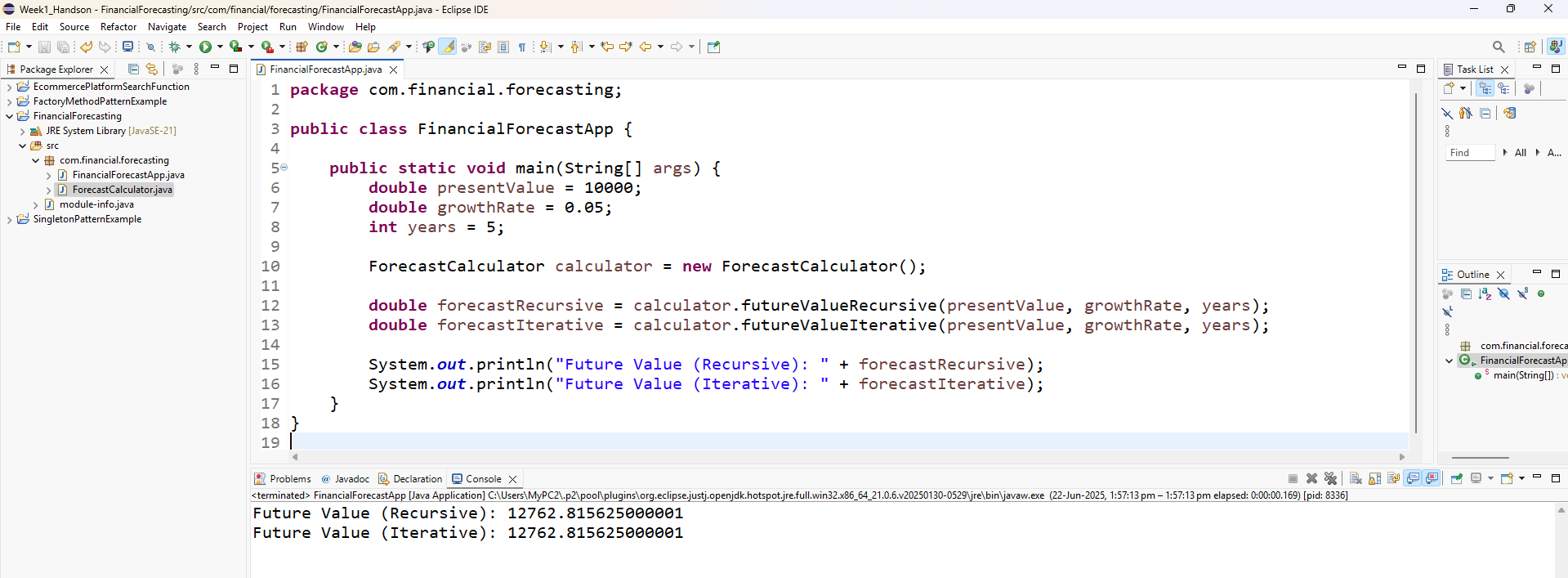
}

return pv;

}

}

**Output:**

****

**Time Complexity of the Recursive Algorithm**

public double futureValueRecursive(double pv, double rate, int years) {

if (years == 0) {

return pv;

}

return futureValueRecursive(pv \* (1 + rate), rate, years - 1);

}

**Analysis:**

* Each recursive call reduces years by 1.
* The function performs a **single multiplication and recursive call** each time.

So, the total number of recursive calls is equal to n (i.e., the number of years).

**Time Complexity:**

* **O(n)** — Linear time complexity

**Optimization to Avoid Excessive Computation**

Although the recursion is simple and linear, it still has some downsides in Java:

**1. Recursive Stack Overhead**

* Java uses a call stack to manage recursion.
* Too many recursive calls (e.g., years = 10,000) may cause a **StackOverflowError**.

**2. Not Tail-Recursive in Java**

* This version is **not tail-optimized**, and Java doesn't optimize tail recursion like functional languages.

**3. Convert to Iterative Method (Best Practice in Java)**

java

CopyEdit

public double futureValueIterative(double pv, double rate, int years) {

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

pv \*= (1 + rate);

}

return pv;

}

* **Time Complexity:** O(n)
* **Space Complexity:** **O(1)** (No recursion stack)
* Safe for large values of n

**4. Use Formula Directly (Constant Time)**

Mathematically, the future value is:

FV=PV×(1+r)nFV = PV \times (1 + r)^nFV=PV×(1+r)n

java

CopyEdit

public double futureValueFormula(double pv, double rate, int years) {

return pv \* Math.pow(1 + rate, years);

}

* **Time Complexity:** **O(1)** (handled internally by Math.pow)
* **Most Efficient** for financial forecasting