**DATA STRUCTURES AND ALGORITHMS**

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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**SOLUTION:**

**BIG O NOTATION:**

**Big O notation** is used to **describe the efficiency** of an algorithm in terms of **time and space complexity** as the input size grows.

O(1)- super fast

O(n) -time expands as input grows

O(log n)- very efficient

**Best, average, and worst-case scenarios for search operations:**

**Best case -**  The minimum time the algorithm will take.

**Average case** - The expected time for the input.

**Worst case** - The maximum time the algorithm takes.

**LINEAR SEARCH**

**Best case -** O(1) Element present in the first.

**Average case** - o(n/2)- Element present in the middle.

**Worst case -** o(n)- Element is at last or not present.

**BINARY SEARCH**

**Best case -** O(1) Element present in middle.

**Average case** - o(log n)

**Worst case -** o(log n) - keeps halving until one element is present.

**IMPLEMENTATION**

Product.java

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;  
 }  
  
 public String toString() {  
 return "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;  
 }  
}

LinearSearch.java

public class LinearSearch {  
 public static Product linearSearch(Product[] products, String name) {  
 for (Product product : products) {  
 if (product.productName.equalsIgnoreCase(name)) {  
 return product;  
 }  
 }  
 return null;  
 }  
}

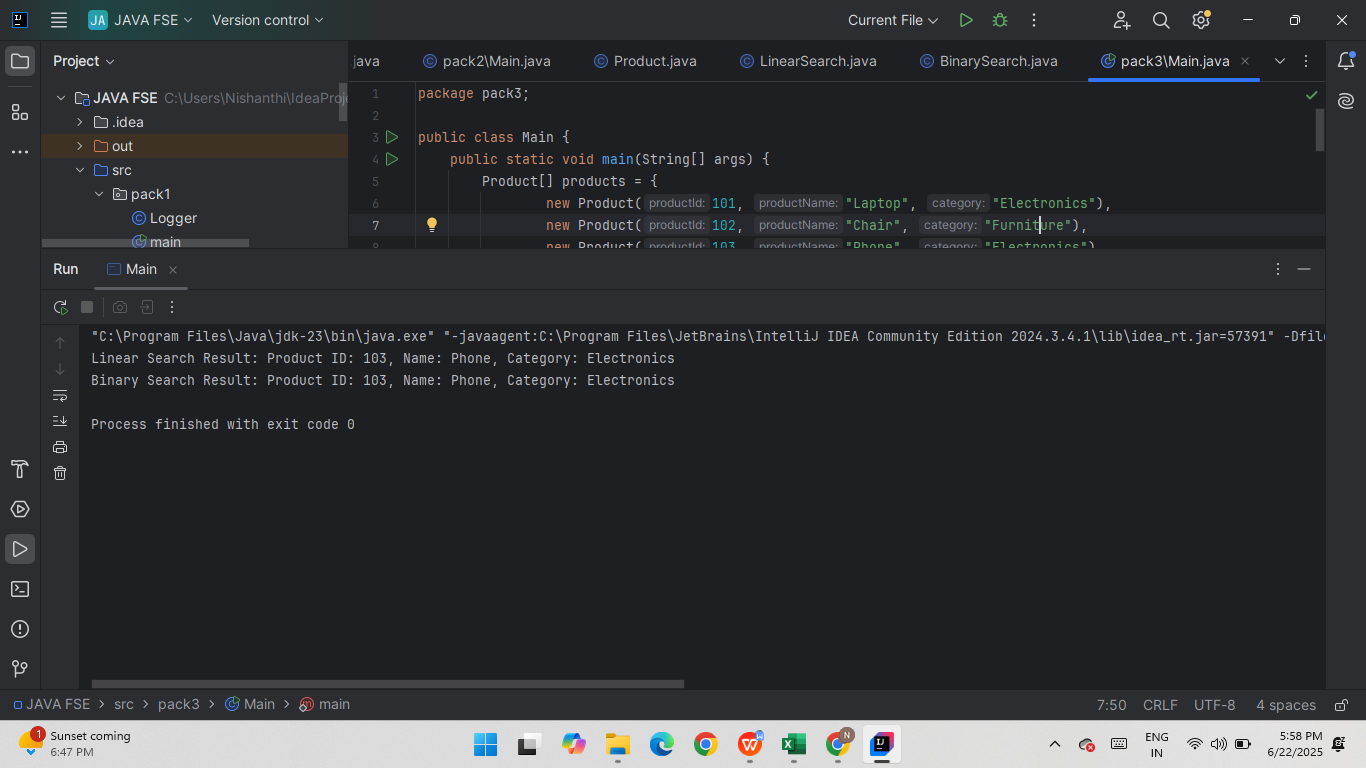
BinarySearch.java

public class BinarySearch {  
  
 // bubble sort  
 public static void sortByName(Product[] products) {  
 for (int i = 0; i < products.length - 1; i++) {  
 for (int j = 0; j < products.length - i - 1; j++) {  
 if (products[j].productName.compareToIgnoreCase(products[j + 1].productName) > 0) {  
 Product temp = products[j];  
 products[j] = products[j + 1];  
 products[j + 1] = temp;  
 }  
 }  
 }  
 }  
  
 // Binary search   
 public static Product search(Product[] products, String name) {  
 *sortByName*(products);   
  
 int left = 0;  
 int right = products.length - 1;  
  
 while (left <= right) {  
 int mid = (left + right) / 2;  
 int result = products[mid].productName.compareToIgnoreCase(name);  
  
 if (result == 0) {  
 return products[mid];  
 } else if (result < 0) {  
 left = mid + 1;  
 } else {  
 right = mid - 1;  
 }  
 }  
 return null;  
 }  
}

Main.java

public class Main {  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(101, "Laptop", "Electronics"),  
 new Product(102, "Chair", "Furniture"),  
 new Product(103, "Phone", "Electronics"),  
 new Product(104, "Book", "Education")  
 };  
  
 Product foundLinear = LinearSearch.*linearSearch*(products, "Phone");  
 System.*out*.println("Linear Search Result: " + (foundLinear != null ? foundLinear : "Not Found"));  
  
 Product foundBinary = BinarySearch.*search*(products, "Phone");  
 System.*out*.println("Binary Search Result: " + (foundBinary != null ? foundBinary : "Not Found"));  
 }  
}

OUTPUT:



**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**SOLUTION:**

**RECURSION:**

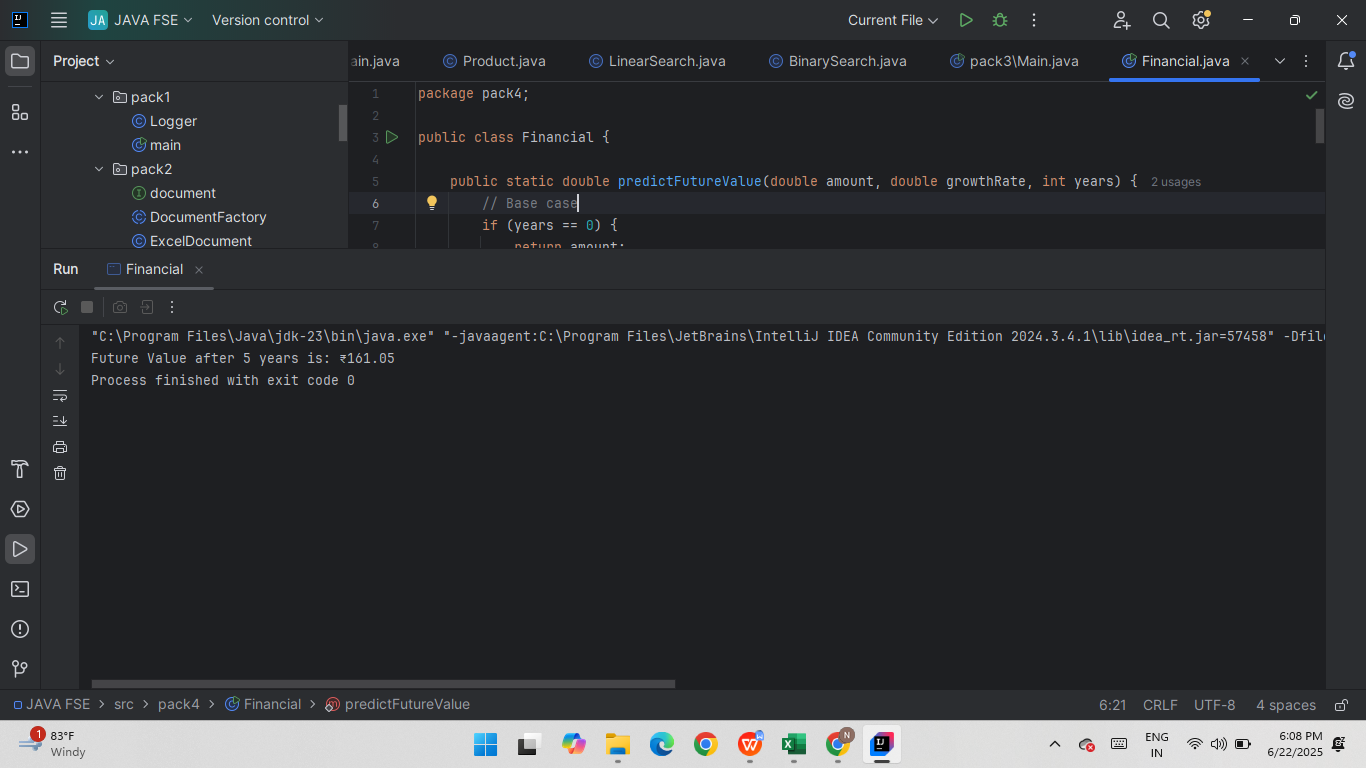
Recursion is a concept where the function calls itself. It simplifies problem that follow repeated patterns.

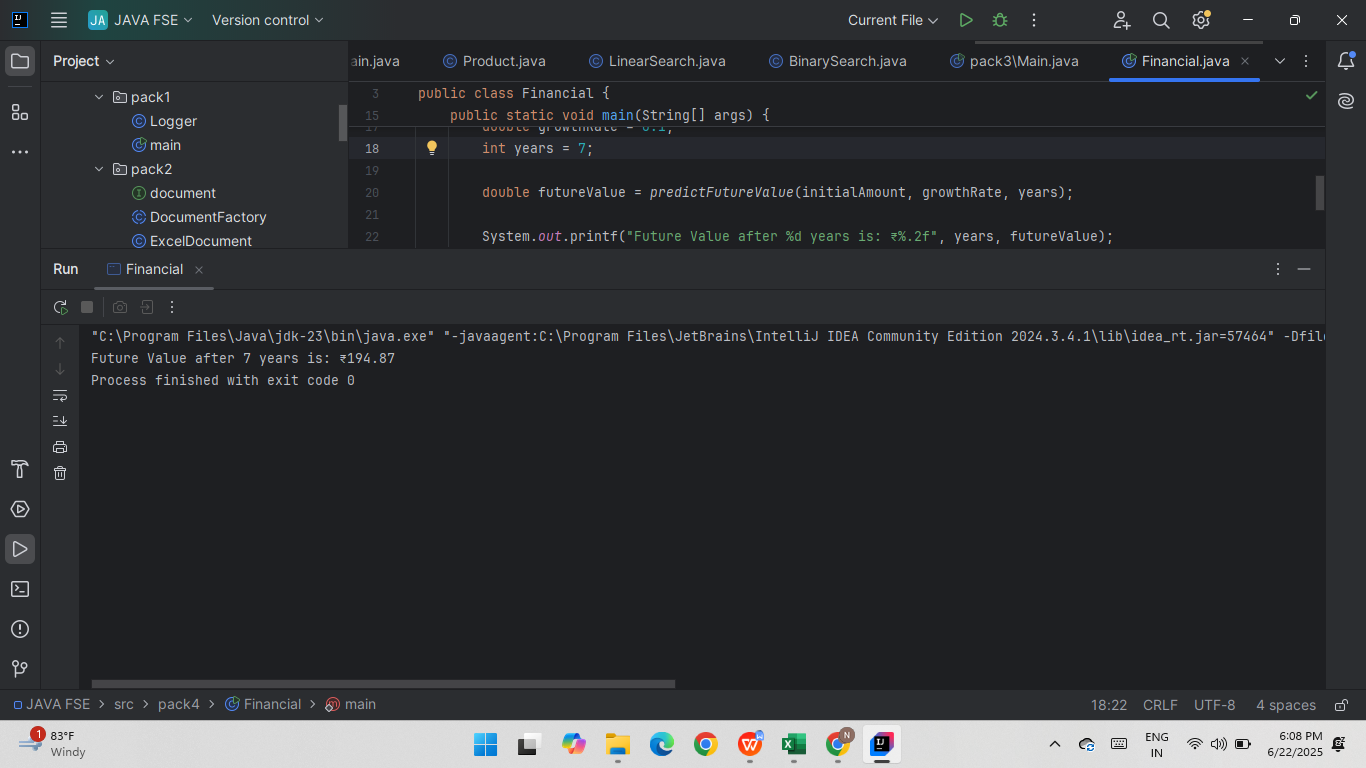
**IMPLEMENTATION:**

Financial.java

public class Financial {  
  
 public static double predictFutureValue(double amount, double growthRate, int years) {  
 // Base case  
 if (years == 0) {  
 return amount;  
 }  
  
 double previousValue = *predictFutureValue*(amount, growthRate, years - 1);  
 return previousValue \* (1 + growthRate);  
 }  
  
 public static void main(String[] args) {  
 double initialAmount = 100;  
 double growthRate = 0.1;  
 int years = 5;  
  
 double futureValue = *predictFutureValue*(initialAmount, growthRate, years);  
  
 System.*out*.printf("Future Value after %d years is: ₹%.2f", years, futureValue);  
 }  
}

OUTPUT:





**Time Complexity:**

Each recursive call decreases yaers by 1.

The total number of calls = n (number of years)

**Time Complexity: O(n)**

**OPTIMIZATION:**

Can be optimized using **memoization**  for more complex problems.

**Memoization:**

Saving the results of previous calculations in a **temporary storage (like an array)** so we don’t have to calculate the same thing again and again.