WEEK - 1

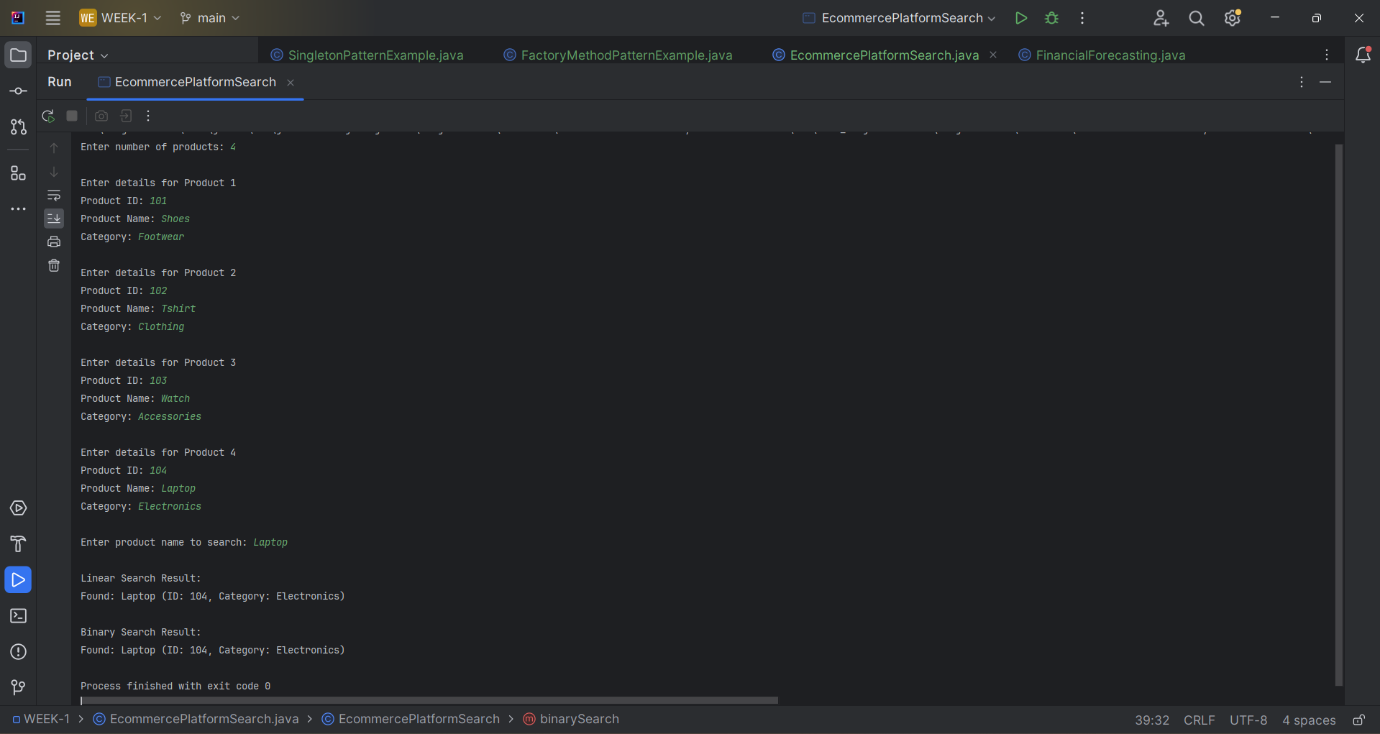
Algorithms Data Structures

Exercise – 2 : E-commerce Platform Search Function

Code :

import java.util.\*;  
  
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 productName + " (ID: " + productId + ", Category: " + category + ")";  
 }  
}  
  
public class EcommercePlatformSearch {  
  
 // Linear search scans each product one by one and compares the name  
 // Time Complexity: Best - O(1), Average - O(n), Worst - O(n)  
 public static Product linearSearch(List<Product> products, String targetName) {  
 for (Product product : products) {  
 if (product.productName.equalsIgnoreCase(targetName)) {  
 return product;  
 }  
 }  
 return null;  
 }  
  
 // Binary search works on a sorted list and repeatedly divides the search range  
 // Time Complexity: Best - O(1), Average/Worst - O(log n) but requires sorted input  
 public static Product binarySearch(List<Product> products, String targetName) {  
 int left = 0;  
 int right = products.size() - 1;  
  
 while (left <= right) {  
 int mid = left + (right - left) / 2;  
 Product midProduct = products.get(mid);  
 int cmp = midProduct.productName.compareToIgnoreCase(targetName);  
  
 if (cmp == 0) {  
 return midProduct;  
 } else if (cmp < 0) {  
 left = mid + 1;  
 } else {  
 right = mid - 1;  
 }  
 }  
 return null;  
 }  
  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 List<Product> inventory = new ArrayList<>();  
  
 System.*out*.print("Enter number of products: ");  
 int n = scanner.nextInt();  
 scanner.nextLine(); // Consume newline  
  
 // Collect product details from user  
 for (int i = 0; i < n; i++) {  
 System.*out*.println("\nEnter details for Product " + (i + 1));  
 System.*out*.print("Product ID: ");  
 int id = scanner.nextInt();  
 scanner.nextLine();  
 System.*out*.print("Product Name: ");  
 String name = scanner.nextLine();  
 System.*out*.print("Category: ");  
 String category = scanner.nextLine();  
  
 inventory.add(new Product(id, name, category));  
 }  
  
 System.*out*.print("\nEnter product name to search: ");  
 String target = scanner.nextLine();  
  
 // Perform linear search on the unsorted list  
 Product foundLinear = *linearSearch*(inventory, target);  
 System.*out*.println("\nLinear Search Result:");  
 System.*out*.println(foundLinear != null ? "Found: " + foundLinear : "Product not found.");  
  
 // Sort the list by product name before binary search  
 inventory.sort(Comparator.*comparing*(p -> p.productName.toLowerCase()));  
  
 // Perform binary search on the sorted list  
 Product foundBinary = *binarySearch*(inventory, target);  
 System.*out*.println("\nBinary Search Result:");  
 System.*out*.println(foundBinary != null ? "Found: " + foundBinary : "Product not found.");  
  
 /\*  
 Time Complexity Comparison:  
 Linear Search:  
 - Best Case: O(1) // if the product is at the start  
 - Average Case: O(n) // on average, we may check half the list  
 - Worst Case: O(n) // if the product is at the end or not present  
  
 Binary Search:  
 - Best Case: O(1) // if the product is in the middle  
 - Average Case: O(log n)  
 - Worst Case: O(log n)  
 - Requires sorted data to work correctly  
 \*/  
  
 /\*  
 Algorithm Suitability:  
 - Linear search is simple and works without any sorting, but is inefficient for large datasets.  
 - Binary search is much faster, especially for large inventories, but it requires the list to be sorted.  
 - For an actual e-commerce platform where product count is high, binary search is preferred due to better performance.  
 - In real-world applications, advanced indexing (like hash maps or search trees) is even better, but for this assignment,  
 binary search gives a strong balance between speed and implementation complexity.  
 \*/  
  
 scanner.close();  
 }  
}

Output :



Exercise – 7 : Financial Forecasting

Code :

import java.util.Scanner;  
  
public class FinancialForecasting {  
  
 // Recursive function to calculate future value  
 // Formula: futureValue(n) = futureValue(n-1) \* (1 + rate)  
 public static double forecastFutureValue(double initialValue, double annualGrowthRate, int years) {  
 if (years == 0) {  
 return initialValue; // Base case: Year 0, return starting value  
 }  
 return *forecastFutureValue*(initialValue, annualGrowthRate, years - 1) \* (1 + annualGrowthRate);  
 }  
  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
  
 // Input: Initial amount, growth rate, number of years  
 System.*out*.print("Enter initial investment amount: ");  
 double initial = scanner.nextDouble();  
  
 System.*out*.print("Enter expected annual growth rate (e.g., 0.05 for 5%): ");  
 double rate = scanner.nextDouble();  
  
 System.*out*.print("Enter number of years to forecast: ");  
 int years = scanner.nextInt();  
  
 double futureValue = *forecastFutureValue*(initial, rate, years);  
  
 System.*out*.printf("Predicted future value after %d years: ₹%.2f%n", years, futureValue);  
  
 /\*  
 Time Complexity Analysis:  
 The recursive function runs once for each year.  
 Therefore, Time Complexity = O(n), where n = number of years.  
  
 There is no redundant computation here, so memoization is not necessary.  
 However, for more complex or overlapping recursive problems,  
 dynamic programming or memoization would be used to avoid recomputation.  
 \*/  
  
 /\*  
 Optimization Tips:  
 - Recursion is clean for small input (like 10-100 years), but for very large input, prefer iteration.  
 - Iterative methods are often faster and use less stack memory.  
 - If the problem had overlapping subproblems, memoization (storing already computed values)  
 would help reduce time complexity from exponential to linear.  
 \*/  
  
 scanner.close();  
 }  
}

Output :

