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

Product:

package 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;

}

public String toString() {

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

}

}

**Main class**:

ProductSearchDemo:

package search;

import java.util.Arrays;

import java.util.Comparator;

public class ProductSearchDemo {

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

for (Product product : products) {

if (product.productId == targetId) {

return product;

}

}

return null;

}

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

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

while (low <= high) {

int mid = (low + high) / 2;

if (products[mid].productId == targetId)

return products[mid];

else if (products[mid].productId < targetId)

low = mid + 1;

else

high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] productList = {

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

new Product(205, "Laptop", "Electronics"),

new Product(150, "Shirt", "Clothing"),

new Product(120, "Watch", "Accessories"),

};

Product result1 = linearSearch(productList, 205);

System.out.println("Linear Search Result: " + (result1 != null ? result1 : "Product not found"));

Arrays.sort(productList, Comparator.comparingInt(p -> p.productId));

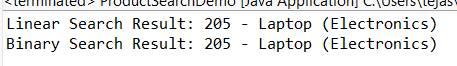
Product result2 = binarySearch(productList, 205);

System.out.println("Binary Search Result: " + (result2 != null ? result2 : "Product not found"));

}

}

Output:



**Exercise 7: Financial Forecasting:**

FinancialForecast:

public class FinancialForecast {

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

if (years == 0) {

return currentValue;

}

double newValue = currentValue \* (1 + growthRate);

return predictFutureValue(newValue, growthRate, years - 1);

}

public static double predictFutureValueMemo(double currentValue, double growthRate, int years, double[] memo) {

if (years == 0) return currentValue;

if (memo[years] != 0) return memo[years];

memo[years] = predictFutureValueMemo(currentValue, growthRate, years - 1, memo) \* (1 + growthRate);

return memo[years];

}

public static void main(String[] args) {

double initialValue = 1000.0;

double annualGrowthRate = 0.05;

int years = 4;

double forecastedValue = predictFutureValue(initialValue, annualGrowthRate, years);

System.out.printf("Predicted Value after %d years (recursive): %.2f%n", years, forecastedValue);

double[] memo = new double[years + 1];

double forecastedMemo = predictFutureValueMemo(initialValue, annualGrowthRate, years, memo);

System.out.printf("Predicted Value after %d years (memoized): %.2f%n", years, forecastedMemo);

System.out.println("\n=== Analysis ===");

System.out.println("Recursion simplifies problems like compound growth by expressing them in terms of smaller subproblems.");

System.out.println("Time Complexity of naive recursion: O(n), where n is the number of years.");

System.out.println("Memoization helps avoid redundant computations in overlapping recursive calls.");

System.out.println("For this problem, since each year depends only on the previous year, memoization isn't strictly needed but useful in more complex models.");

}

}

Output:

