E-commerce Platform Search Function

Explanation on Asymptotic Notation:

\* Big O Notation:

\* Describes the time complexity of an algorithm as input size grows.

\* Linear Search: O(n)

\* Each element is checked one by one.

\* Binary Search: O(log n)

\* Works on sorted arrays; repeatedly divides the search range in half.

\* Best / Average / Worst Case:

\* Linear Search: Best O(1), Average O(n/2), Worst O(n)

\* Binary Search: Best O(1), Average/Worst O(log n)

Product.java

package ECommerceSearch;

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;

    }

    @Override

    public String toString() {

        return "[" + productId + "] " + productName + " (" + category + ")";

    }

}

SearchUtil.java

package ECommerceSearch;

public class SearchUtils {

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

        for (Product p : products) {

            if (p.productId == targetId) {

                return p;

            }

        }

        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;

    }

}

Main.java

package ECommerceSearch;

import java.util.Arrays;

import java.util.Comparator;

public class Main {

    public static void main(String[] args) {

        Product[] products = {

            new Product(102, "Phone", "Electronics"),

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

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

            new Product(100, "Watch", "Accessories")

        };

        System.out.println("=== Linear Search ===");

        Product result1 = SearchUtils.linearSearch(products, 103);

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

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

        System.out.println("\n=== Binary Search ===");

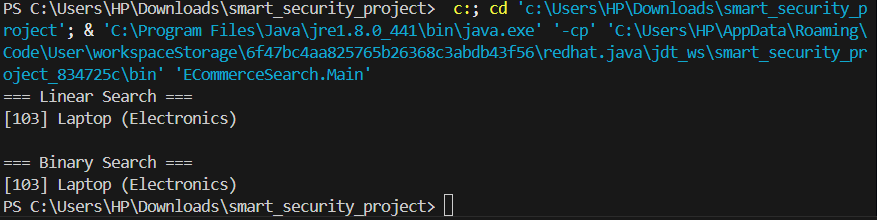
        Product result2 = SearchUtils.binarySearch(products, 103);

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

    }

}

Output:



* To implement financial forecasting

Concept of recursion:

\* Recursion:

\* A method that calls itself to solve smaller instances of a problem.

\* It simplifies problems that have repeating patterns, such as financial projections.

\* Example:

Future Value = Present Value × (1 + growthRate)^years

\* Recursively: FV(n) = FV(n-1) × (1 + r)

Financialforecasting.java

package FinancialForecasting;

public class FinancialForecast {

    // Recursive method to calculate future value

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

        if (years == 0) return currentValue;

        return forecastRecursive(currentValue \* (1 + growthRate), growthRate, years - 1);

    }

    // Optimized with memoization

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

        if (years == 0) return currentValue;

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

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

        return memo[years];

    }

    public static void main(String[] args) {

        double initialValue = 1000.0;

        double growthRate = 0.10;

        int forecastYears = 5;

        // Run basic recursion

        double futureValue = forecastRecursive(initialValue, growthRate, forecastYears);

        System.out.println("Future Value (recursive): " + futureValue);

        // Run memoized version

        Double[] memo = new Double[forecastYears + 1];

        double futureMemo = forecastMemo(initialValue, growthRate, forecastYears, memo);

        System.out.println("Future Value (memoized): " + futureMemo);

    }

}

Output:

