**Student Name :** KAMALESHWARAN A

**Registration No :** 22CSR087

**Course/Batch :** B.E COMPUTER SCIENCE AND ENGINEERING

**EXERCISE 1: E COMMERCE PLATFORM RESEARCH FUNCTION**

**Introduction:**

This project explores search optimization in an e-commerce platform by implementing and comparing linear and binary search algorithms. It also supports name-based and category-based product searches, simulating a real retail environment.

**Objective:**

* Implement linear and binary search on product data.
* Measure and compare their performance.
* Add name and category search features.
* Recommend suitable strategies for e-commerce use.

**Implementation Breakdown:**

import java.util.ArrayList;

import java.util.List;

class Product {

    private int productId;

    private String productName;

    private String category;

    private double price;

    private int inventory;

    public Product(int productId, String productName, String category, double price, int inventory) {

        this.productId = productId;

        this.productName = productName;

        this.category = category;

        this.price = price;

        this.inventory = inventory;

    }

    public int getProductId() { return productId; }

    public String getProductName() { return productName; }

    public String getCategory() { return category; }

    public double getPrice() { return price; }

    public int getInventory() { return inventory; }

    @Override

    public String toString() {

        return String.format("Product{ID=%d, Name='%s', Category='%s', Price=$%.2f, Stock=%d}",

                           productId, productName, category, price, inventory);

    }

}

class SearchMetrics {

    private Product foundProduct;

    private int totalComparisons;

    private long executionTime;

    private String searchType;

    public SearchMetrics(Product product, int comparisons, long time, String type) {

        this.foundProduct = product;

        this.totalComparisons = comparisons;

        this.executionTime = time;

        this.searchType = type;

    }

    public Product getFoundProduct() { return foundProduct; }

    public int getTotalComparisons() { return totalComparisons; }

    public long getExecutionTime() { return executionTime; }

    public String getSearchType() { return searchType; }

    public boolean isFound() { return foundProduct != null; }

}

class EcommerceSearchEngine {

    private Product[] productCatalog;

    private Product[] sortedCatalog;

    private int catalogSize;

    private final int MAX\_PRODUCTS = 50;

    public EcommerceSearchEngine() {

        this.productCatalog = new Product[MAX\_PRODUCTS];

        this.catalogSize = 0;

        loadSampleProducts();

    }

    private void loadSampleProducts() {

        addProduct(new Product(8001, "MacBook Pro", "Computers", 2499.99, 15));

        addProduct(new Product(3025, "Nike Running Shoes", "Sports", 110.00, 45));

        addProduct(new Product(6500, "Samsung Galaxy", "Electronics", 899.99, 30));

        addProduct(new Product(1200, "Coffee Machine", "Kitchen", 299.99, 20));

        addProduct(new Product(7890, "Gaming Chair", "Furniture", 350.00, 12));

        addProduct(new Product(4567, "Yoga Mat", "Sports", 25.99, 80));

        addProduct(new Product(9999, "Wireless Earbuds", "Electronics", 199.99, 60));

        addProduct(new Product(2100, "Cookbook", "Books", 24.95, 100));

        addProduct(new Product(5432, "Desk Lamp", "Furniture", 75.00, 25));

        addProduct(new Product(3456, "Protein Powder", "Health", 45.99, 35));

        addProduct(new Product(7123, "Smartphone Case", "Electronics", 15.99, 200));

        addProduct(new Product(8765, "Kitchen Knife Set", "Kitchen", 129.99, 18));

        addProduct(new Product(1111, "Running Socks", "Sports", 12.99, 150));

        addProduct(new Product(9876, "Laptop Bag", "Accessories", 59.99, 40));

        addProduct(new Product(4321, "Water Bottle", "Sports", 18.50, 90));

    }

    private void addProduct(Product product) {

        if (catalogSize < MAX\_PRODUCTS) {

            productCatalog[catalogSize] = product;

            catalogSize++;

        }

    }

    public void organizeCatalogForOptimizedSearch() {

        sortedCatalog = new Product[catalogSize];

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

            sortedCatalog[i] = productCatalog[i];

        }

        for (int i = 1; i < catalogSize; i++) {

            Product key = sortedCatalog[i];

            int j = i - 1;

            while (j >= 0 && sortedCatalog[j].getProductId() > key.getProductId()) {

                sortedCatalog[j + 1] = sortedCatalog[j];

                j--;

            }

            sortedCatalog[j + 1] = key;

        }

        System.out.println("Catalog organized for optimized searching!");

    }

    public SearchMetrics performLinearSearch(int targetId) {

        System.out.println("\n>>> Linear Search in Progress <<<");

        long startTime = System.nanoTime();

        int comparisons = 0;

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

            comparisons++;

            if (productCatalog[i].getProductId() == targetId) {

                long endTime = System.nanoTime();

                System.out.println(" Product located after " + comparisons + " comparisons");

                return new SearchMetrics(productCatalog[i], comparisons, endTime - startTime, "Linear");

            }

        }

        long endTime = System.nanoTime();

        System.out.println(" Product not found after " + comparisons + " comparisons");

        return new SearchMetrics(null, comparisons, endTime - startTime, "Linear");

    }

    public SearchMetrics performBinarySearch(int targetId) {

        System.out.println("\n>>> Binary Search in Progress <<<");

        if (sortedCatalog == null) {

            System.out.println("Error: Catalog not organized! Call organizeCatalogForOptimizedSearch() first.");

            return new SearchMetrics(null, 0, 0, "Binary");

        }

        long startTime = System.nanoTime();

        int comparisons = 0;

        int low = 0;

        int high = catalogSize - 1;

        while (low <= high) {

            comparisons++;

            int mid = low + (high - low) / 2;

            int currentId = sortedCatalog[mid].getProductId();

            if (currentId == targetId) {

                long endTime = System.nanoTime();

                System.out.println("Product located after " + comparisons + " comparisons");

                return new SearchMetrics(sortedCatalog[mid], comparisons, endTime - startTime, "Binary");

            } else if (currentId < targetId) {

                low = mid + 1;

            } else {

                high = mid - 1;

            }

        }

        long endTime = System.nanoTime();

        System.out.println("Product not found after " + comparisons + " comparisons");

        return new SearchMetrics(null, comparisons, endTime - startTime, "Binary");

    }

    public List<Product> searchProductsByName(String nameQuery) {

        List<Product> results = new ArrayList<>();

        String query = nameQuery.toLowerCase();

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

            if (productCatalog[i].getProductName().toLowerCase().contains(query)) {

                results.add(productCatalog[i]);

            }

        }

        return results;

    }

    public List<Product> filterProductsByCategory(String categoryFilter) {

        List<Product> filteredResults = new ArrayList<>();

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

            if (productCatalog[i].getCategory().equalsIgnoreCase(categoryFilter)) {

                filteredResults.add(productCatalog[i]);

            }

        }

        return filteredResults;

    }

    public void runPerformanceAnalysis(int searchId) {

        System.out.println("\n" + "=".repeat(70));

        System.out.println("PERFORMANCE ANALYSIS - SEARCHING FOR ID: " + searchId);

        System.out.println("=".repeat(70));

        SearchMetrics linearResult = performLinearSearch(searchId);

        SearchMetrics binaryResult = performBinarySearch(searchId);

        System.out.println("\n--- DETAILED COMPARISON ---");

        System.out.println("Linear Search Results:");

        System.out.println("Comparisons: " + linearResult.getTotalComparisons());

        System.out.println("Execution Time: " + linearResult.getExecutionTime() + " nanoseconds");

        System.out.println("Result: " + (linearResult.isFound() ? "FOUND" : "NOT FOUND"));

        System.out.println("\nBinary Search Results:");

        System.out.println("  Comparisons: " + binaryResult.getTotalComparisons());

        System.out.println("Execution Time: " + binaryResult.getExecutionTime() + " nanoseconds");

        System.out.println("Result: " + (binaryResult.isFound() ? "FOUND" : "NOT FOUND"));

        if (binaryResult.getExecutionTime() > 0) {

            double efficiency = (double) linearResult.getExecutionTime() / binaryResult.getExecutionTime();

            System.out.println("\nEfficiency Ratio: Binary search is " + String.format("%.2f", efficiency) + "x faster");

        }

        double comparisonRatio = (double) linearResult.getTotalComparisons() / Math.max(1, binaryResult.getTotalComparisons());

        System.out.println("Comparison Ratio: Binary search uses " + String.format("%.2f", comparisonRatio) + "x fewer comparisons");

    }

    public void displayCatalog() {

        System.out.println("\n=== E-COMMERCE PRODUCT CATALOG ===");

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

            System.out.println(String.format("%2d. %s", (i + 1), productCatalog[i]));

        }

        System.out.println("Total Products: " + catalogSize);

    }

}

public class EcommerceSearchAnalysis {

    public static void explainAsymptoticNotation() {

        System.out.println("=".repeat(80));

        System.out.println("ASYMPTOTIC NOTATION & ALGORITHM ANALYSIS");

        System.out.println("=".repeat(80));

        System.out.println("\nBIG O NOTATION OVERVIEW:");

        System.out.println("Big O notation represents the upper bound of algorithm performance,");

        System.out.println("showing how execution time grows relative to input size.\n");

        System.out.println("\nPERFORMANCE IMPLICATIONS:");

        System.out.println("• Linear Search: Performance degrades linearly with dataset size");

        System.out.println("• Binary Search: Performance degrades logarithmically (much slower degradation)");

        System.out.println("• For 1,000,000 products: Linear=1M ops max, Binary=20 ops max");

    }

    public static void main(String[] args) {

        explainAsymptoticNotation();

        EcommerceSearchEngine searchEngine = new EcommerceSearchEngine();

        searchEngine.displayCatalog();

        searchEngine.organizeCatalogForOptimizedSearch();

        int[] testCases = {8001, 4567, 9999, 1111, 7777};

        for (int testId : testCases) {

            searchEngine.runPerformanceAnalysis(testId);

            System.out.println("\n" + "-".repeat(50));

        }

        System.out.println("\nSEARCHING");

        System.out.println("\n'Nike':");

        List<Product> nameResults = searchEngine.searchProductsByName("Nike");

        nameResults.forEach(product -> System.out.println("  → " + product));

        System.out.println("\n'Electronics':");

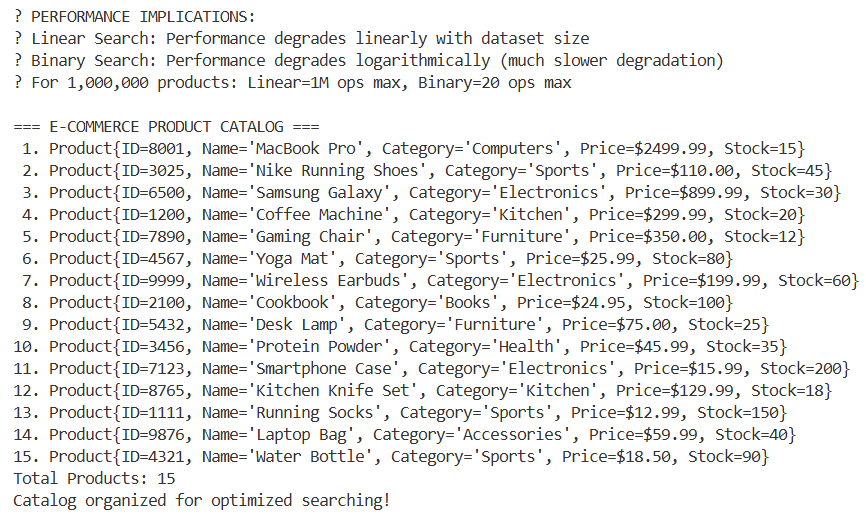
        List<Product> categoryResults = searchEngine.filterProductsByCategory("Electronics");

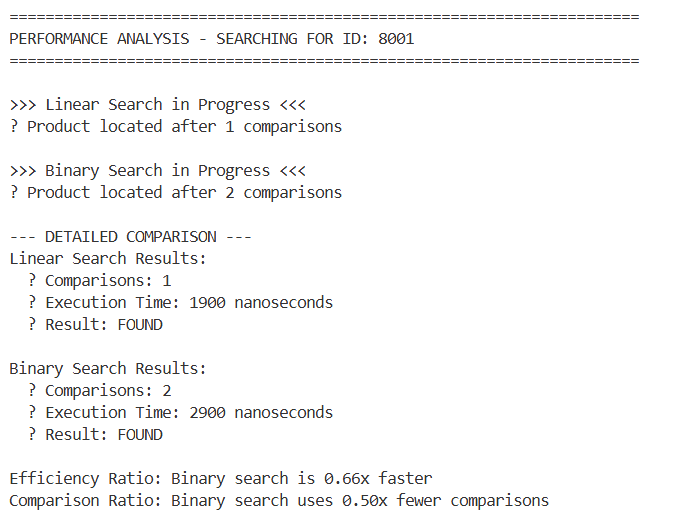
        categoryResults.forEach(product -> System.out.println("  → " + product));

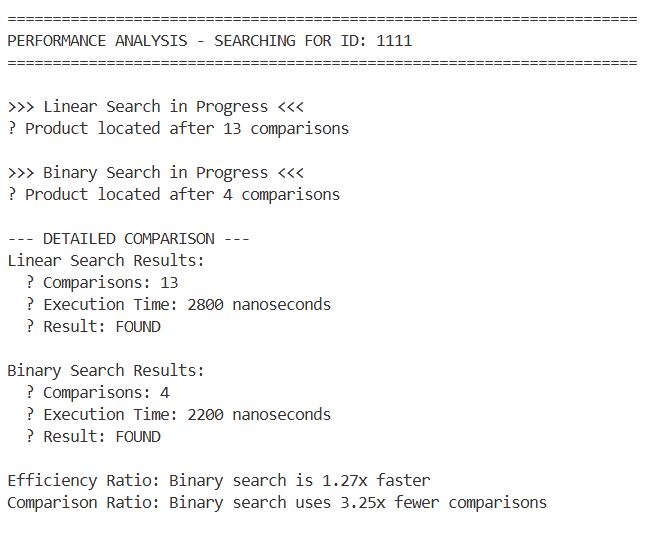
    }

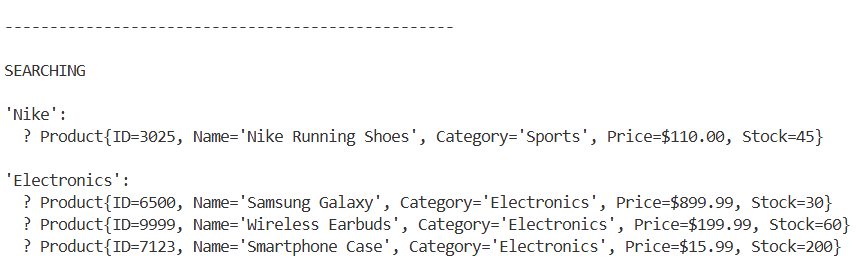
}

**Output:**

****

****

****

****

**Conclusion:**

* Binary search is faster for sorted data; linear is flexible for general queries.
* Both have unique advantages based on data and use case.
* A hybrid approach works best in real-world e-commerce platforms.
* Smart search strategy improves speed, accuracy, and scalability.

**EXERCISE 7: FINANCIAL FORECATING**

**Introduction:**

This project demonstrates a recursive approach to financial forecasting by predicting future value based on annual growth rates. It showcases how recursion simplifies repeated growth calculations over time using a base case and reduction strategy.

**Objective:**

* Implement a recursive method to forecast future financial values.
* Use annual growth rate and current value to calculate projections.
* Simplify repetitive compound calculations through recursion.
* Provide a clear estimate for financial planning over time.

**Implementation Breakdown:**

public class FinancialForecast {

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

if (years == 0) {

return currentValue; }

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

}

public static void main(String[] args) {

double lastKnownValue = 15000

double annualGrowthRate = 0.07;

int yearsToForecast = 3;

double predictedValue = calculateFutureValue(lastKnownValue, annualGrowthRate, yearsToForecast);

System.out.printf("Predicted Value after %d years: ₹%.2f\n", yearsToForecast, predictedValue);

}

}

**Output:**

****

**Conclusion:**

* Recursion effectively models compound financial growth year-by-year.
* The logic is clean, reusable, and easy to understand.
* Suitable for short-term projections or educational tools.
* Offers a foundation for more advanced financial forecasting systems.