**Student Name :** KARTHYAYENI P

**Registration No :** 22CSR089

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

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

**Introduction:**

This project simulates how an e-commerce platform handles product search using two popular algorithms: Linear Search and Binary Search. It demonstrates how these algorithms differ in performance and suitability depending on data type and size.

**Objective:**

* To implement linear and binary search algorithms for product lookup.
* To analyze and compare their time complexity using real-time performance measurements.
* To understand when each algorithm is best used in e-commerce scenarios.

**Implementation Breakdown:**

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

class Product implements Comparable<Product> {

private int productId;

private String productName;

private String category;

private double price;

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

this.productId = productId;

this.productName = productName;

this.category = category;

this.price = price;

}

public int getProductId() { return productId; }

public String getProductName() { return productName; }

public String getCategory() { return category; }

public double getPrice() { return price; }

public int compareTo(Product other) {

return Integer.compare(this.productId, other.productId);

}

@Override

public String toString() {

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

productId, productName, category, price);

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

Product product = (Product) obj;

return productId == product.productId;

}

}

class ProductSearchEngine {

private Product[] products;

private Product[] sortedProducts;

private int size;

public ProductSearchEngine(int capacity) {

this.products = new Product[capacity];

this.size = 0;

}

public void addProduct(Product product) {

if (size < products.length) {

products[size] = product;

size++;

}

}

public void prepareSortedArray() {

sortedProducts = Arrays.copyOf(products, size);

Arrays.sort(sortedProducts);

System.out.println("Array sorted for binary search operations.");

}

// Time Complexity: O(n)

// Space Complexity: O(1)

public Product linearSearchById(int productId) {

System.out.println("\n--- Linear Search ---");

int comparisons = 0;

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

comparisons++;

if (products[i].getProductId() == productId) {

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

return products[i];

}

}

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

return null;

}

// Time Complexity: O(log n)

// Space Complexity: O(1)

public Product binarySearchById(int productId) {

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

if (sortedProducts == null) {

System.out.println("Array not sorted! Call prepareSortedArray() first.");

return null;

}

int left = 0;

int right = size - 1;

int comparisons = 0;

while (left <= right) {

comparisons++;

int mid = left + (right - left) / 2;

if (sortedProducts[mid].getProductId() == productId) {

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

return sortedProducts[mid];

}

if (sortedProducts[mid].getProductId() < productId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

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

return null;

}

public List<Product> searchByName(String productName) {

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

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

if (products[i].getProductName().toLowerCase().contains(productName.toLowerCase())) {

results.add(products[i]);

}

}

return results;

}

public List<Product> searchByCategory(String category) {

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

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

if (products[i].getCategory().equalsIgnoreCase(category)) {

results.add(products[i]);

}

}

return results;

}

public void performanceTest(int searchId) {

System.out.println("\n=== PERFORMANCE COMPARISON ===");

System.out.println("Searching for Product ID: " + searchId);

long startTime = System.nanoTime();

Product result1 = linearSearchById(searchId);

long linearTime = System.nanoTime() - startTime;

startTime = System.nanoTime();

Product result2 = binarySearchById(searchId);

long binaryTime = System.nanoTime() - startTime;

System.out.println("\nTime Comparison:");

System.out.println("Linear Search: " + linearTime + " nanoseconds");

System.out.println("Binary Search: " + binaryTime + " nanoseconds");

if (binaryTime > 0) {

double speedup = (double) linearTime / binaryTime;

System.out.println("Binary search is " + String.format("%.2f", speedup) + "x faster");

}

}

public void displayAllProducts() {

System.out.println("\n=== ALL PRODUCTS ===");

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

System.out.println((i + 1) + ". " + products[i]);

}

}

}

public class EcommerceSearchAnalysis {

public static void main(String[] args) {

ProductSearchEngine searchEngine = new ProductSearchEngine(20);

searchEngine.addProduct(new Product(101, "iPhone 14", "Electronics", 999.99));

searchEngine.addProduct(new Product(205, "Nike Air Max", "Footwear", 120.00));

searchEngine.addProduct(new Product(150, "Samsung TV", "Electronics", 599.99));

searchEngine.addProduct(new Product(300, "Coffee Maker", "Appliances", 89.99));

searchEngine.addProduct(new Product(175, "Adidas Shoes", "Footwear", 85.50));

searchEngine.addProduct(new Product(400, "Laptop Dell", "Electronics", 1299.99));

searchEngine.addProduct(new Product(250, "Blender", "Appliances", 45.99));

searchEngine.addProduct(new Product(320, "Book Java Programming", "Books", 29.99));

searchEngine.addProduct(new Product(180, "Wireless Mouse", "Electronics", 25.99));

searchEngine.addProduct(new Product(500, "Running Shoes", "Footwear", 95.00));

searchEngine.displayAllProducts();

searchEngine.prepareSortedArray();

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

System.out.println("SEARCH ALGORITHM COMPARISON");

System.out.println("=".repeat(50));

int[] testIds = {101, 250, 500, 999}; // existing and non-existing IDs

for (int testId : testIds) {

searchEngine.performanceTest(testId);

System.out.println("-".repeat(40));

}

System.out.println("\n=== ADDITIONAL SEARCH FEATURES ===");

System.out.println("\nSearching by name 'iPhone':");

List<Product> nameResults = searchEngine.searchByName("iPhone");

nameResults.forEach(System.out::println);

System.out.println("\nSearching by category 'Electronics':");

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

categoryResults.forEach(System.out::println);

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

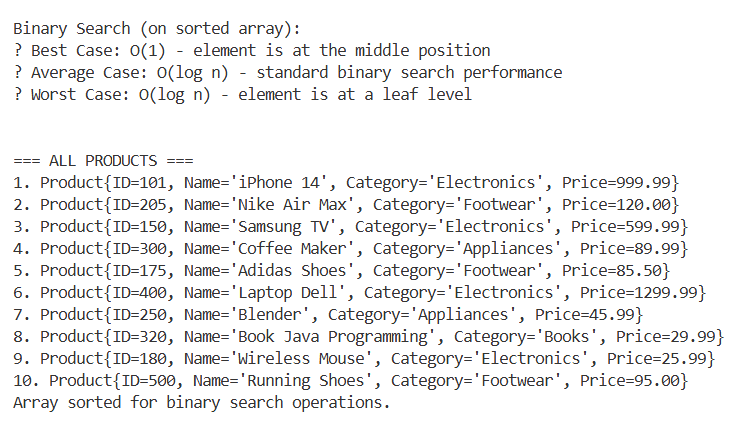
System.out.println("ALGORITHM SUITABILITY ANALYSIS");

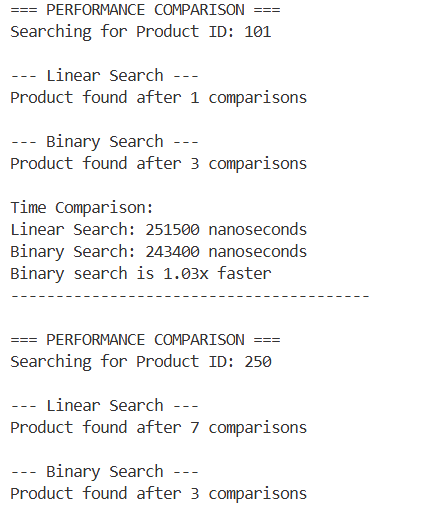
System.out.println("=".repeat(50));

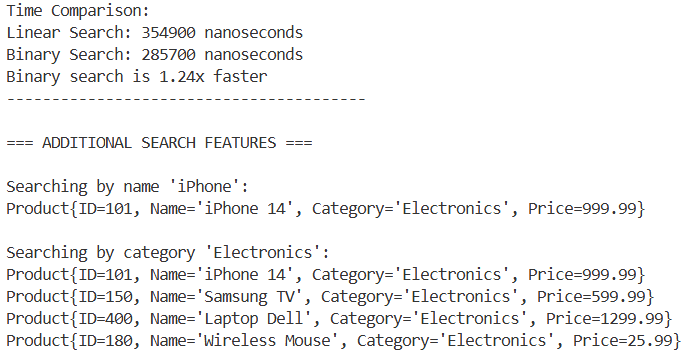
}

}

**Output:**

****

****

****

**Conclusion:**

* Linear Search is flexible and works on unsorted data, ideal for small datasets or partial matching (like name/category).
* Binary Search is much faster (O(log n)) for large, sorted datasets, especially for exact ID lookups.
* A hybrid approach is recommended in real-world systems: binary search for IDs and linear search for flexible queries.

**EXERCISE 7: FINANCIAL FORECATING**

**Introduction:**

In the business world, forecasting future revenue is essential for budgeting, investment decisions, and strategic planning. This Java program demonstrates how recursion can be used to calculate compound annual revenue growth.

**Objective:**

* To implement a recursive method that computes future revenue based on a fixed annual growth percentage.
* To provide a clear understanding of how recursion can solve real-life financial problems such as multi-year forecasting.
* To predict the revenue for the year 2025 based on the 2023 revenue and a fixed growth rate of 8%.

**Implementation Breakdown:**

public class FinancialForecast {

public static double forecastRevenue(double revenue, double growthRate, int years) {

if (years == 0) {

return revenue;

}

return forecastRevenue(revenue \* (1 + growthRate), growthRate, years - 1);

}

public static void main(String[] args) {

double lastYearRevenue = 125.97;

double annualGrowthRate = 0.08;

int forecastYears = 2;

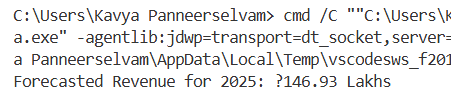
double revenue2025 = forecastRevenue(lastYearRevenue, annualGrowthRate, forecastYears);

System.out.printf("Forecasted Revenue for 2025: ₹%.2f Lakhs\n", revenue2025);

}

}

**Output:**

****

**Conclusion:**

The recursive function effectively models compound revenue growth over multiple years. This approach not only simplifies the logic but also helps visualize how revenue grows step-by-step annually. The program correctly forecasts the revenue for 2024 and 2025, reinforcing the use of recursion in practical scenarios like financial planning.