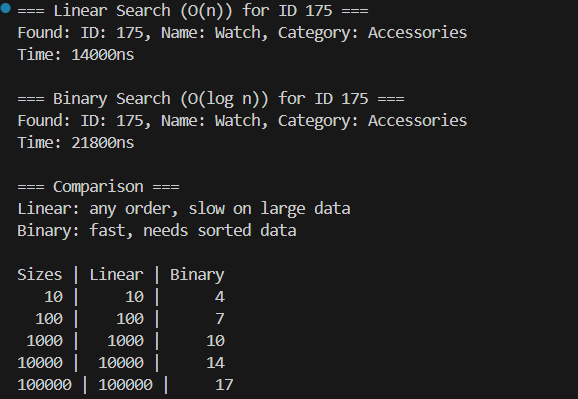
# 1. EcommerceSearchExample.java

**/\*  
 \* BIG O NOTATION - SIMPLE EXPLANATION:  
 \*   
 \* Big O tells us how slow an algorithm gets when data size increases  
 \*   
 \* Common Big O types:  
 \* O(1) - Always same speed (best)  
 \* O(log n) - Gets slower very slowly (very good)   
 \* O(n) - Gets slower at same rate as data grows (okay)  
 \* O(n²) - Gets much slower quickly (bad)  
 \*   
 \* SEARCH SCENARIOS:  
 \* Best case - Find item immediately (lucky)  
 \* Average case - Find item in middle (normal)  
 \* Worst case - Find item at end or not found (unlucky)  
 \*   
 \* LINEAR SEARCH: O(n) - check each item one by one  
 \* BINARY SEARCH: O(log n) - split data in half repeatedly (much faster)  
 \*/**  
import java.util.Arrays;  
  
class Product {  
 int id;  
 String name, category;  
  
 Product(int id, String name, String category) {  
 this.id = id;  
 this.name = name;  
 this.category = category;  
 }  
  
 public String toString() {  
 return String.format("ID: %d, Name: %s, Category: %s", id, name, category);  
 }  
}  
  
public class EcommerceSearchExample {  
 static Product[] products, sortedProducts;  
  
 public static void main(String[] args) {  
 setupProducts();  
 searchAndReport(products, 175, false);  
 searchAndReport(sortedProducts, 175, true);  
 analyzeSearches();  
 }  
  
 static void setupProducts() {  
 products = new Product[] {  
 new Product(150, "Laptop", "Electronics"),  
 new Product(101, "Phone", "Electronics"),  
 new Product(205, "Shirt", "Clothing"),  
 new Product(300, "Book", "Education"),  
 new Product(175, "Watch", "Accessories"),  
 new Product(250, "Shoes", "Clothing"),  
 new Product(125, "Tablet", "Electronics"),  
 new Product(400, "Bag", "Accessories")  
 };  
 sortedProducts = products.clone();  
 Arrays.sort(sortedProducts, (a, b) -> a.id - b.id);  
 }  
  
 static void searchAndReport(Product[] arr, int targetId, boolean isBinary) {  
 String type = isBinary ? "Binary" : "Linear";  
 System.out.printf("=== %s Search (O(%s)) for ID %d ===\n", type, isBinary ? "log n" : "n", targetId);  
 long start = System.nanoTime();  
 Product result = isBinary ? binarySearch(arr, targetId) : linearSearch(arr, targetId);  
 long time = System.nanoTime() - start;  
 System.out.println(result != null ? "Found: " + result : "Not found");  
 System.out.println("Time: " + time + "ns\n");  
 }  
  
 static Product linearSearch(Product[] arr, int target) {  
 for (int i = 0; i < arr.length; i++) {  
 if (arr[i].id == target)  
 return arr[i];  
 }  
 return null;  
 }  
  
 static Product binarySearch(Product[] arr, int target) {  
 int l = 0, r = arr.length - 1;  
 while (l <= r) {  
 int m = (l + r) >>> 1;  
 if (arr[m].id == target)  
 return arr[m];  
 if (arr[m].id < target)  
 l = m + 1;  
 else  
 r = m - 1;  
 }  
 return null;  
 }  
  
 static void analyzeSearches() {  
 System.out.println("=== Comparison ===");  
 System.out.println("Linear: any order, slow on large data");  
 System.out.println("Binary: fast, needs sorted data\n");  
 System.out.println("Sizes | Linear | Binary");  
 int[] sizes = { 10, 100, 1000, 10000, 100000 };  
 for (int n : sizes) {  
 System.out.printf("%5d | %6d | %6d\n", n, n, (int) Math.ceil(Math.log(n) / Math.log(2)));  
 }  
 }  
}

Output :



# 2. FinancialForecastingExample.java

**/\*  
 \* RECURSION - SIMPLE EXPLANATION:  
 \*   
 \* Recursion = function calls itself to solve smaller problems  
 \* Like counting down: 5, 4, 3, 2, 1, STOP!  
 \*   
 \* Needs:  
 \* 1. Base case - when to stop  
 \* 2. Recursive case - call itself with smaller input  
 \*   
 \* FINANCIAL FORECASTING:  
 \* Calculate investment growth year by year using recursion  
 \*/**  
import java.util.HashMap;  
import java.util.Map;  
  
public class FinancialForecastingExample {  
 static Map<String, Double> cache = new HashMap<>();  
 static int calls = 0;  
   
 public static void main(String[] args) {  
 System.out.println("=== FINANCIAL FORECASTING ===\n");  
   
 testBasicRecursion();  
 testWithMemoization();  
 showComplexity();  
 }  
   
 static void testBasicRecursion() {  
 System.out.println("BASIC RECURSION:");  
 double money = 1000.0;  
 double rate = 0.10; // 10% growth  
 int years = 3;  
   
 System.out.println("Initial: $" + money);  
 System.out.println("Growth: " + (rate \* 100) + "% per year");  
 System.out.println("Years: " + years + "\n");  
   
 calls = 0;  
 double result = calculateFuture(money, rate, years);  
   
 System.out.println("Final amount: $" + String.format("%.2f", result));  
 System.out.println("Recursive calls: " + calls + "\n");  
 }  
   
 static double calculateFuture(double amount, double rate, int years) {  
 calls++;  
 System.out.println("Year " + years + ": $" + String.format("%.2f", amount));  
   
 if (years == 0) {  
 return amount; // Base case - stop here  
 }  
   
 // Recursive case - grow money and reduce years  
 return calculateFuture(amount \* (1 + rate), rate, years - 1);  
 }  
   
 static void testWithMemoization() {  
 System.out.println("WITH MEMOIZATION (CACHING):");  
 cache.clear();  
 calls = 0;  
   
 double result = calculateFutureMemo(1000.0, 0.10, 3);  
 System.out.println("Final amount: $" + String.format("%.2f", result));  
 System.out.println("Calls with cache: " + calls);  
 System.out.println("Cache size: " + cache.size() + "\n");  
 }  
   
 static double calculateFutureMemo(double amount, double rate, int years) {  
 calls++;  
 String key = amount + "\_" + years;  
   
 if (cache.containsKey(key)) {  
 System.out.println("Cache hit for year " + years);  
 return cache.get(key);  
 }  
   
 double result;  
 if (years == 0) {  
 result = amount;  
 } else {  
 result = calculateFutureMemo(amount \* (1 + rate), rate, years - 1);  
 }  
   
 cache.put(key, result);  
 return result;  
 }  
   
 static void showComplexity() {  
 System.out.println("=== COMPLEXITY ANALYSIS ===");  
   
 System.out.println("RECURSIVE APPROACH:");  
 System.out.println("Time: O(n) - n recursive calls");  
 System.out.println("Space: O(n) - call stack memory");  
   
 System.out.println("\nMEMOIZED APPROACH:");  
 System.out.println("Time: O(n) first time, O(1) if cached");  
 System.out.println("Space: O(n) - cache + call stack");  
   
 System.out.println("\nITERATIVE (BEST FOR THIS PROBLEM):");  
 double result = calculateIterative(1000.0, 0.10, 3);  
 System.out.println("Result: $" + String.format("%.2f", result));  
 System.out.println("Time: O(n), Space: O(1)");  
   
 System.out.println("\nWHEN TO USE RECURSION:");  
 System.out.println("✓ Tree/graph problems");  
 System.out.println("✓ Divide and conquer");  
 System.out.println("✗ Simple counting (use loops instead)");  
 }  
   
 static double calculateIterative(double amount, double rate, int years) {  
 System.out.println("Iterative calculation:");  
 for (int i = 0; i < years; i++) {  
 amount \*= (1 + rate);  
 System.out.println("Year " + (i + 1) + ": $" + String.format("%.2f", amount));  
 }  
 return amount;  
 }  
}

Output:

A screenshot of a computer program

AI-generated content may be incorrect.