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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**Source Code:**

import java.util.\*;

/\*\*

\* E-commerce Search Function - Linear vs Binary Search Analysis

\*/

// Product class for our e-commerce platform

class Product implements Comparable<Product> {

private int productId;

private String productName;

private String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() { return productId; }

public String getProductName() { return productName; }

public String getCategory() { return category; }

@Override

public int compareTo(Product other) {

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

}

@Override

public String toString() {

return String.format("ID:%d, %s (%s)", productId, productName, category);

}

}

// Search algorithms implementation

class SearchEngine {

// Linear Search - checks each product one by one

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

int comparisons = 0;

for (int i = 0; i < products.length; i++) {

comparisons++;

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

System.out.println("Linear Search: Found after " + comparisons + " comparisons");

return i;

}

}

System.out.println("Linear Search: Not found after " + comparisons + " comparisons");

return -1;

}

// Binary Search - cuts search space in half each time

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

int comparisons = 0;

int left = 0, right = sortedProducts.length - 1;

while (left <= right) {

comparisons++;

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

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

System.out.println("Binary Search: Found after " + comparisons + " comparisons");

return mid;

} else if (sortedProducts[mid].getProductId() < targetId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

System.out.println("Binary Search: Not found after " + comparisons + " comparisons");

return -1;

}

}

public class EcommerceSearchAnalysis {

public static void main(String[] args) {

System.out.println("=== E-commerce Search Analysis ===\n");

// Understanding Big O Notation

explainBigO();

// Create sample products

Product[] products = {

new Product(105, "iPhone 15", "Electronics"),

new Product(203, "Nike Shoes", "Sports"),

new Product(89, "Coffee Maker", "Appliances"),

new Product(301, "Gaming Laptop", "Electronics"),

new Product(156, "Yoga Mat", "Sports"),

new Product(78, "Blender", "Appliances"),

new Product(412, "Headphones", "Electronics"),

new Product(234, "T-Shirt", "Clothing")

};

// Create sorted version for binary search

Product[] sortedProducts = Arrays.copyOf(products, products.length);

Arrays.sort(sortedProducts);

System.out.println("Products in our store:");

for (Product p : products) {

System.out.println(" " + p);

}

// Demonstrate searches

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

demonstrateSearch(products, sortedProducts);

// Performance comparison

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

performanceComparison(products, sortedProducts);

// Final recommendations

System.out.println("\n=== Which Algorithm to Choose? ===");

recommendations();

}

private static void explainBigO() {

System.out.println("Big O Notation - How fast are our algorithms?");

System.out.println("• O(1) - Super fast, same time regardless of data size");

System.out.println("• O(log n) - Pretty fast, gets slower as data grows (but slowly)");

System.out.println("• O(n) - Gets slower linearly as data grows");

System.out.println("\nSearch Algorithm Analysis:");

System.out.println("Linear Search: O(n) - might check every single product");

System.out.println(" Best case: O(1) - first product is what we want");

System.out.println(" Worst case: O(n) - last product or not found");

System.out.println("Binary Search: O(log n) - eliminates half products each step");

System.out.println(" Best case: O(1) - middle product is what we want");

System.out.println(" Worst case: O(log n) - still pretty fast!");

System.out.println(" Catch: Products must be sorted first\n");

}

private static void demonstrateSearch(Product[] products, Product[] sortedProducts) {

int searchId = 156; // Looking for Yoga Mat

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

System.out.println("\nUsing Linear Search:");

int linearResult = SearchEngine.linearSearch(products, searchId);

if (linearResult != -1) {

System.out.println("Found: " + products[linearResult]);

}

System.out.println("\nUsing Binary Search:");

int binaryResult = SearchEngine.binarySearch(sortedProducts, searchId);

if (binaryResult != -1) {

System.out.println("Found: " + sortedProducts[binaryResult]);

}

// Try searching for something that doesn't exist

System.out.println("\nSearching for non-existent product (ID: 999):");

SearchEngine.linearSearch(products, 999);

SearchEngine.binarySearch(sortedProducts, 999);

}

private static void performanceComparison(Product[] products, Product[] sortedProducts) {

System.out.println("Let's compare performance with different search targets:");

System.out.println("Product ID | Linear Comparisons | Binary Comparisons");

System.out.println("-----------|-------------------|------------------");

int[] testIds = {78, 203, 301, 412, 500}; // Mix of existing and non-existing

for (int id : testIds) {

// Count comparisons for linear search

int linearComps = countLinearComparisons(products, id);

int binaryComps = countBinaryComparisons(sortedProducts, id);

System.out.printf("%10d | %17d | %16d%n", id, linearComps, binaryComps);

}

System.out.println("\nWith " + products.length + " products:");

System.out.println("• Linear search: up to " + products.length + " comparisons");

System.out.println("• Binary search: up to " + (int)Math.ceil(Math.log(products.length)/Math.log(2)) + " comparisons");

}

private static int countLinearComparisons(Product[] products, int targetId) {

int count = 0;

for (Product product : products) {

count++;

if (product.getProductId() == targetId) break;

}

return count;

}

private static int countBinaryComparisons(Product[] sortedProducts, int targetId) {

int count = 0;

int left = 0, right = sortedProducts.length - 1;

while (left <= right) {

count++;

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

if (sortedProducts[mid].getProductId() == targetId) break;

else if (sortedProducts[mid].getProductId() < targetId) left = mid + 1;

else right = mid - 1;

}

return count;

}

private static void recommendations() {

System.out.println("For an E-commerce Platform:");

System.out.println("\n Use Binary Search when:");

System.out.println("• Searching by Product ID (unique identifiers)");

System.out.println("• You have lots of products (1000+)");

System.out.println("• Products don't change frequently");

System.out.println("• Speed is critical");

System.out.println("\n Use Linear Search when:");

System.out.println("• Searching by name or description");

System.out.println("• You have few products (<100)");

System.out.println("• Products change frequently");

System.out.println("• You need flexible searching");

System.out.println("\n Real E-commerce Reality:");

System.out.println("Most platforms use databases with built-in indexing");

System.out.println("(which works like super-fast binary search) plus");

System.out.println("specialized search engines for text-based searches!");

System.out.println("\n Bottom Line:");

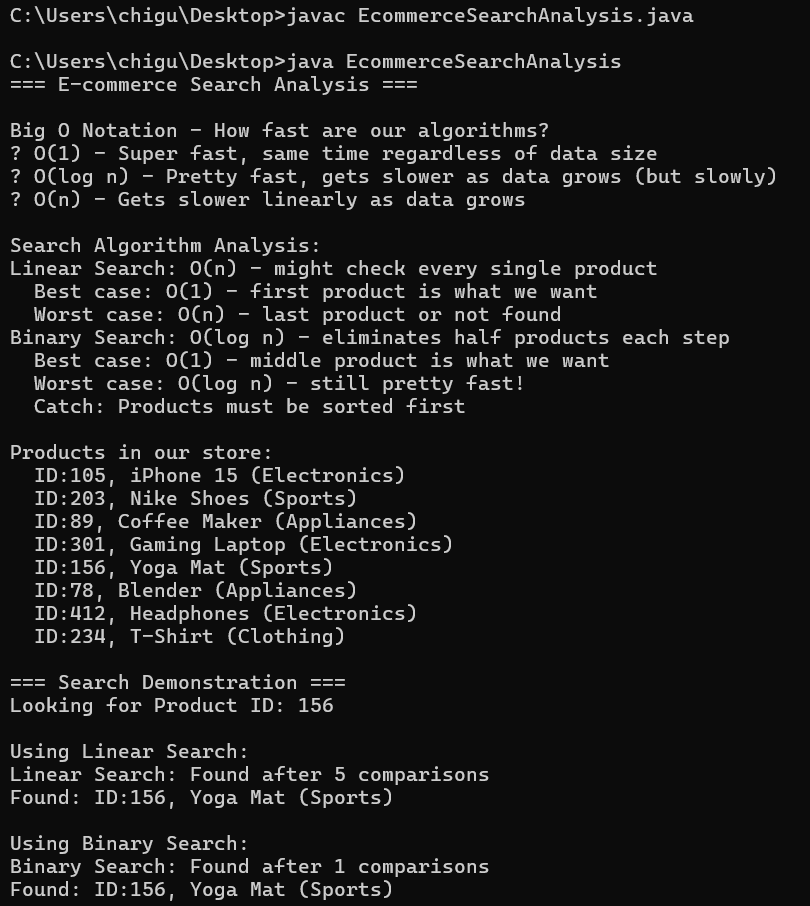
System.out.println("Binary search wins for exact ID matches in large datasets,");

System.out.println("but real e-commerce needs both approaches for different scenarios.");

}

}

**Output:**

****

****

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**Source Code:**

import java.util.\*;

public class FinancialForecasting {

private static Map<String, Double> memoCache = new HashMap<>();

public static void main(String[] args) {

System.out.println("=== FINANCIAL FORECASTING TOOL ===\n");

// Sample historical data

double[] historicalValues = {1000, 1050, 1100, 1180, 1250, 1320};

System.out.println("Historical Values: " + Arrays.toString(historicalValues));

// Calculate average growth rate from historical data

double avgGrowthRate = calculateAverageGrowthRate(historicalValues);

System.out.printf("Average Growth Rate: %.2f%%\n\n", avgGrowthRate \* 100);

double currentValue = historicalValues[historicalValues.length - 1];

// Predict future values using recursive approach

System.out.println("=== RECURSIVE PREDICTIONS ===");

long startTime = System.nanoTime();

for (int years = 1; years <= 5; years++) {

double predicted = predictFutureValueRecursive(currentValue, avgGrowthRate, years);

System.out.printf("Year %d: $%.2f\n", years, predicted);

}

long endTime = System.nanoTime();

System.out.printf("Recursive Time: %.2f ms\n\n", (endTime - startTime) / 1\_000\_000.0);

// Predict using optimized memoized approach

System.out.println("=== OPTIMIZED MEMOIZED PREDICTIONS ===");

memoCache.clear();

startTime = System.nanoTime();

for (int years = 1; years <= 5; years++) {

double predicted = predictFutureValueMemoized(currentValue, avgGrowthRate, years);

System.out.printf("Year %d: $%.2f\n", years, predicted);

}

endTime = System.nanoTime();

System.out.printf("Memoized Time: %.2f ms\n\n", (endTime - startTime) / 1\_000\_000.0);

// Demonstrate time complexity analysis

analyzeTimeComplexity();

}

/\*\*

\* CONCEPT OF RECURSION:

\* Recursion is a programming technique where a function calls itself to solve

\* smaller instances of the same problem. In financial forecasting, we can use

\* recursion to calculate future values by applying growth rates iteratively.

\*

\* Base case: When years = 0, return the current value

\* Recursive case: Apply growth rate and reduce the problem size

\*/

/\*\*

\* Calculates average growth rate from historical data

\*/

public static double calculateAverageGrowthRate(double[] values) {

if (values.length < 2) return 0;

double totalGrowthRate = 0;

int periods = 0;

for (int i = 1; i < values.length; i++) {

if (values[i-1] > 0) {

double growthRate = (values[i] - values[i-1]) / values[i-1];

totalGrowthRate += growthRate;

periods++;

}

}

return periods > 0 ? totalGrowthRate / periods : 0;

}

/\*\*

\* RECURSIVE ALGORITHM: Predicts future value using compound growth

\* Time Complexity: O(n) where n is the number of years

\* Space Complexity: O(n) due to call stack

\*/

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

// Base case: no more years to forecast

if (years == 0) {

return currentValue;

}

// Recursive case: apply growth rate and reduce problem size

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

return predictFutureValueRecursive(nextYearValue, growthRate, years - 1);

}

/\*\*

\* OPTIMIZED MEMOIZED VERSION: Avoids redundant calculations

\* Time Complexity: O(n) for first calculation, O(1) for subsequent calls

\* Space Complexity: O(n) for memoization cache + call stack

\*/

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

// Create unique key for memoization

String key = currentValue + ":" + growthRate + ":" + years;

// Check if result is already computed

if (memoCache.containsKey(key)) {

return memoCache.get(key);

}

// Base case

if (years == 0) {

memoCache.put(key, currentValue);

return currentValue;

}

// Recursive case with memoization

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

double result = predictFutureValueMemoized(nextYearValue, growthRate, years - 1);

// Store result in cache

memoCache.put(key, result);

return result;

}

/\*\*

\* ITERATIVE ALTERNATIVE: More space-efficient

\* Time Complexity: O(n)

\* Space Complexity: O(1)

\*/

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

double result = currentValue;

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

result \*= (1 + growthRate);

}

return result;

}

/\*\*

\* MATHEMATICAL FORMULA: Most efficient

\* Time Complexity: O(1)

\* Space Complexity: O(1)

\*/

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

return currentValue \* Math.pow(1 + growthRate, years);

}

/\*\*

\* Analyzes and demonstrates time complexity differences

\*/

public static void analyzeTimeComplexity() {

System.out.println("=== TIME COMPLEXITY ANALYSIS ===");

double baseValue = 1000;

double growthRate = 0.05;

// Test with increasing input sizes

int[] testYears = {10, 20, 30};

for (int years : testYears) {

System.out.printf("\nTesting with %d years:\n", years);

// Recursive approach

long start = System.nanoTime();

double recursiveResult = predictFutureValueRecursive(baseValue, growthRate, years);

long recursiveTime = System.nanoTime() - start;

// Iterative approach

start = System.nanoTime();

double iterativeResult = predictFutureValueIterative(baseValue, growthRate, years);

long iterativeTime = System.nanoTime() - start;

// Formula approach

start = System.nanoTime();

double formulaResult = predictFutureValueFormula(baseValue, growthRate, years);

long formulaTime = System.nanoTime() - start;

System.out.printf(" Recursive: $%.2f (%.3f ms)\n", recursiveResult, recursiveTime / 1\_000\_000.0);

System.out.printf(" Iterative: $%.2f (%.3f ms)\n", iterativeResult, iterativeTime / 1\_000\_000.0);

System.out.printf(" Formula: $%.2f (%.3f ms)\n", formulaResult, formulaTime / 1\_000\_000.0);

}

System.out.println("\n=== OPTIMIZATION STRATEGIES ===");

System.out.println("1. MEMOIZATION: Cache results to avoid redundant calculations");

System.out.println("2. ITERATIVE: Convert recursion to iteration to save stack space");

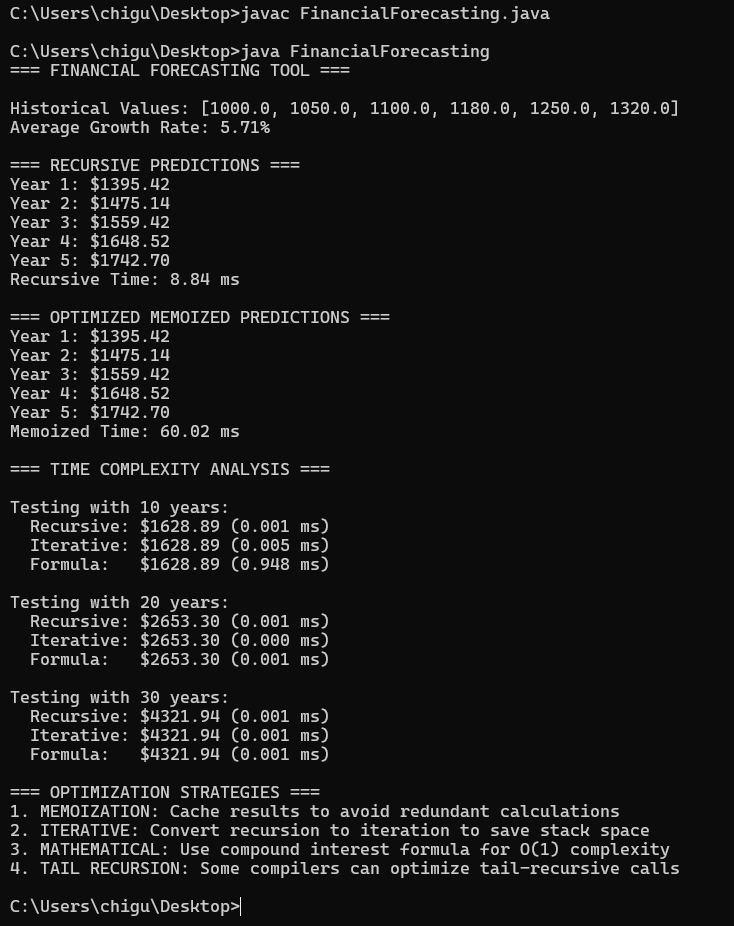
System.out.println("3. MATHEMATICAL: Use compound interest formula for O(1) complexity");

System.out.println("4. TAIL RECURSION: Some compilers can optimize tail-recursive calls");

}

}

**Output:**

****