**P Karthiyayini**

**Superset ID: 6377593**

**Exercise 1: Inventory Management System**

**Scenario:**

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

**CODE:**

import java.util.\*;

// Product class to represent inventory items

class Product {

private String productId;

private String productName;

private int quantity;

private double price;

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

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

// Getters and setters

public String getProductId() { return productId; }

public String getProductName() { return productName; }

public int getQuantity() { return quantity; }

public double getPrice() { return price; }

public void setProductName(String productName) { this.productName = productName; }

public void setQuantity(int quantity) { this.quantity = quantity; }

public void setPrice(double price) { this.price = price; }

@Override

public String toString() {

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

productId, productName, quantity, price);

}

}

// Inventory Management System

class InventoryManager {

// HashMap for O(1) lookup by product ID

private HashMap<String, Product> inventory;

public InventoryManager() {

this.inventory = new HashMap<>();

}

// Add product - O(1) time complexity

public boolean addProduct(Product product) {

if (inventory.containsKey(product.getProductId())) {

System.out.println("Product with ID " + product.getProductId() + " already exists!");

return false;

}

inventory.put(product.getProductId(), product);

System.out.println("Product added successfully: " + product);

return true;

}

// Update product - O(1) time complexity

public boolean updateProduct(String productId, String name, int quantity, double price) {

Product product = inventory.get(productId);

if (product == null) {

System.out.println("Product with ID " + productId + " not found!");

return false;

}

if (name != null) product.setProductName(name);

if (quantity >= 0) product.setQuantity(quantity);

if (price >= 0) product.setPrice(price);

System.out.println("Product updated: " + product);

return true;

}

// Delete product - O(1) time complexity

public boolean deleteProduct(String productId) {

Product removed = inventory.remove(productId);

if (removed == null) {

System.out.println("Product with ID " + productId + " not found!");

return false;

}

System.out.println("Product deleted: " + removed);

return true;

}

// Search product - O(1) time complexity

public Product searchProduct(String productId) {

return inventory.get(productId);

}

// Display all products - O(n) time complexity

public void displayInventory() {

if (inventory.isEmpty()) {

System.out.println("Inventory is empty!");

return;

}

System.out.println("\n=== INVENTORY ===");

for (Product product : inventory.values()) {

System.out.println(product);

}

}

// Get inventory size

public int getInventorySize() {

return inventory.size();

}

// Check low stock items

public List<Product> getLowStockItems(int threshold) {

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

for (Product product : inventory.values()) {

if (product.getQuantity() <= threshold) {

lowStock.add(product);

}

}

return lowStock;

}

}

// Main class to demonstrate the system

public class InventoryManagementSystem {

public static void main(String[] args) {

InventoryManager manager = new InventoryManager();

// Add products

manager.addProduct(new Product("P001", "Laptop", 50, 999.99));

manager.addProduct(new Product("P002", "Mouse", 200, 25.50));

manager.addProduct(new Product("P003", "Keyboard", 150, 75.00));

manager.addProduct(new Product("P004", "Monitor", 30, 299.99));

// Display inventory

manager.displayInventory();

// Update product

manager.updateProduct("P001", null, 45, 949.99);

// Search product

Product found = manager.searchProduct("P002");

System.out.println("\nSearched product: " + found);

// Check low stock

List<Product> lowStock = manager.getLowStockItems(40);

System.out.println("\nLow stock items (≤40): " + lowStock);

// Delete product

manager.deleteProduct("P004");

// Final inventory

manager.displayInventory();

System.out.println("\nTotal products: " + manager.getInventorySize());

}

}

**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.

CODE:

import java.util.\*;

// Product class for e-commerce platform

class Product implements Comparable<Product> {

private String productId;

private String productName;

private String category;

private double price;

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

this.productId = productId;

this.productName = productName;

this.category = category;

this.price = price;

}

// Getters

public String getProductId() { return productId; }

public String getProductName() { return productName; }

public String getCategory() { return category; }

public double getPrice() { return price; }

// For sorting by product name

@Override

public int compareTo(Product other) {

return this.productName.compareToIgnoreCase(other.productName);

}

@Override

public String toString() {

return String.format("Product{ID='%s', 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 Objects.equals(productId, product.productId);

}

@Override

public int hashCode() {

return Objects.hash(productId);

}

}

// Search algorithms implementation

class EcommerceSearchEngine {

private Product[] unsortedProducts; // For linear search

private Product[] sortedProducts; // For binary search

private int size;

public EcommerceSearchEngine(int capacity) {

this.unsortedProducts = new Product[capacity];

this.sortedProducts = new Product[capacity];

this.size = 0;

}

// Add product to both arrays

public void addProduct(Product product) {

if (size >= unsortedProducts.length) {

System.out.println("Storage full! Cannot add more products.");

return;

}

unsortedProducts[size] = product;

sortedProducts[size] = product;

size++;

// Keep sorted array sorted

Arrays.sort(sortedProducts, 0, size);

System.out.println("Added: " + product);

}

// Linear Search - O(n) time complexity

public SearchResult linearSearchByName(String productName) {

long startTime = System.nanoTime();

int comparisons = 0;

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

comparisons++;

if (unsortedProducts[i].getProductName().equalsIgnoreCase(productName)) {

long endTime = System.nanoTime();

return new SearchResult(unsortedProducts[i], i, comparisons, endTime - startTime, "Linear");

}

}

long endTime = System.nanoTime();

return new SearchResult(null, -1, comparisons, endTime - startTime, "Linear");

}

// Binary Search - O(log n) time complexity

public SearchResult binarySearchByName(String productName) {

long startTime = System.nanoTime();

int comparisons = 0;

int left = 0;

int right = size - 1;

while (left <= right) {

comparisons++;

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

int comparison = sortedProducts[mid].getProductName().compareToIgnoreCase(productName);

if (comparison == 0) {

long endTime = System.nanoTime();

return new SearchResult(sortedProducts[mid], mid, comparisons, endTime - startTime, "Binary");

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

long endTime = System.nanoTime();

return new SearchResult(null, -1, comparisons, endTime - startTime, "Binary");

}

// Search by category using linear search

public List<Product> searchByCategory(String category) {

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

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

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

results.add(unsortedProducts[i]);

}

}

return results;

}

// Search by price range

public List<Product> searchByPriceRange(double minPrice, double maxPrice) {

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

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

double price = unsortedProducts[i].getPrice();

if (price >= minPrice && price <= maxPrice) {

results.add(unsortedProducts[i]);

}

}

return results;

}

// Display products

public void displayProducts(boolean sorted) {

Product[] arrayToDisplay = sorted ? sortedProducts : unsortedProducts;

String arrayType = sorted ? "Sorted" : "Unsorted";

System.out.println("\n=== " + arrayType + " Products ===");

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

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

}

}

public int getSize() { return size; }

}

// Search result wrapper

class SearchResult {

private Product product;

private int index;

private int comparisons;

private long timeNanoseconds;

private String algorithm;

public SearchResult(Product product, int index, int comparisons, long timeNanoseconds, String algorithm) {

this.product = product;

this.index = index;

this.comparisons = comparisons;

this.timeNanoseconds = timeNanoseconds;

this.algorithm = algorithm;

}

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

public Product getProduct() { return product; }

public int getComparisons() { return comparisons; }

public double getTimeMicroseconds() { return timeNanoseconds / 1000.0; }

@Override

public String toString() {

if (isFound()) {

return String.format("%s Search: Found %s at index %d (Comparisons: %d, Time: %.2f μs)",

algorithm, product.getProductName(), index, comparisons, getTimeMicroseconds());

} else {

return String.format("%s Search: Not found (Comparisons: %d, Time: %.2f μs)",

algorithm, comparisons, getTimeMicroseconds());

}

}

}

// Performance analyzer

class SearchPerformanceAnalyzer {

public static void compareSearchAlgorithms(EcommerceSearchEngine engine, String searchTerm) {

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

System.out.println("Searching for: '" + searchTerm + "'");

System.out.println("Dataset size: " + engine.getSize() + " products");

// Linear search

SearchResult linearResult = engine.linearSearchByName(searchTerm);

System.out.println(linearResult);

// Binary search

SearchResult binaryResult = engine.binarySearchByName(searchTerm);

System.out.println(binaryResult);

// Performance analysis

if (linearResult.isFound() && binaryResult.isFound()) {

double speedup = linearResult.getTimeMicroseconds() / binaryResult.getTimeMicroseconds();

int comparisonDiff = linearResult.getComparisons() - binaryResult.getComparisons();

System.out.println("\n--- Analysis ---");

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

System.out.println("Binary search used " + comparisonDiff + " fewer comparisons");

}

}

}

// Main demonstration class

public class EcommerceSearchDemo {

public static void main(String[] args) {

EcommerceSearchEngine searchEngine = new EcommerceSearchEngine(20);

// Add sample products

searchEngine.addProduct(new Product("P001", "iPhone 15", "Electronics", 999.99));

searchEngine.addProduct(new Product("P002", "Samsung Galaxy", "Electronics", 899.99));

searchEngine.addProduct(new Product("P003", "Apple MacBook", "Electronics", 1299.99));

searchEngine.addProduct(new Product("P004", "Dell Laptop", "Electronics", 799.99));

searchEngine.addProduct(new Product("P005", "Nike Shoes", "Fashion", 129.99));

searchEngine.addProduct(new Product("P006", "Adidas Shirt", "Fashion", 49.99));

searchEngine.addProduct(new Product("P007", "Coffee Maker", "Home", 89.99));

searchEngine.addProduct(new Product("P008", "Blender", "Home", 59.99));

searchEngine.addProduct(new Product("P009", "Gaming Chair", "Furniture", 299.99));

searchEngine.addProduct(new Product("P010", "Wireless Mouse", "Electronics", 29.99));

// Display products

searchEngine.displayProducts(false);

searchEngine.displayProducts(true);

// Performance comparison

SearchPerformanceAnalyzer.compareSearchAlgorithms(searchEngine, "iPhone 15");

SearchPerformanceAnalyzer.compareSearchAlgorithms(searchEngine, "Gaming Chair");

SearchPerformanceAnalyzer.compareSearchAlgorithms(searchEngine, "NonExistent Product");

// Category search

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

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

System.out.println("Electronics products (" + electronics.size() + " found):");

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

// Price range search

System.out.println("\n=== PRICE RANGE SEARCH ===");

List<Product> affordableProducts = searchEngine.searchByPriceRange(50.0, 200.0);

System.out.println("Products between $50-$200 (" + affordableProducts.size() + " found):");

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

}

}

**Exercise 3: Sorting Customer Orders**

**Scenario:**

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

CODE:

import java.util.\*;

import java.text.SimpleDateFormat;

// Order class representing customer orders

class Order {

private String orderId;

private String customerName;

private double totalPrice;

private Date orderDate;

private String status;

public Order(String orderId, String customerName, double totalPrice, String status) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

this.status = status;

this.orderDate = new Date(); // Current timestamp

}

// Getters

public String getOrderId() { return orderId; }

public String getCustomerName() { return customerName; }

public double getTotalPrice() { return totalPrice; }

public Date getOrderDate() { return orderDate; }

public String getStatus() { return status; }

// Setters

public void setStatus(String status) { this.status = status; }

public void setTotalPrice(double totalPrice) { this.totalPrice = totalPrice; }

@Override

public String toString() {

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm");

return String.format("Order{ID='%s', Customer='%s', Price=$%.2f, Status='%s', Date='%s'}",

orderId, customerName, totalPrice, status, sdf.format(orderDate));

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

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

Order order = (Order) obj;

return Objects.equals(orderId, order.orderId);

}

@Override

public int hashCode() {

return Objects.hash(orderId);

}

}

// Sorting algorithms implementation

class OrderSortingEngine {

// Bubble Sort - O(n²) time complexity

public static SortingResult bubbleSort(Order[] orders) {

long startTime = System.nanoTime();

int comparisons = 0;

int swaps = 0;

int n = orders.length;

for (int i = 0; i < n - 1; i++) {

boolean swapped = false;

for (int j = 0; j < n - i - 1; j++) {

comparisons++;

// Compare by total price (descending order - high value first)

if (orders[j].getTotalPrice() < orders[j + 1].getTotalPrice()) {

// Swap orders

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

swaps++;

swapped = true;

}

}

// If no swapping occurred, array is sorted

if (!swapped) break;

}

long endTime = System.nanoTime();

return new SortingResult("Bubble Sort", comparisons, swaps, endTime - startTime);

}

// Quick Sort - O(n log n) average time complexity

public static SortingResult quickSort(Order[] orders) {

long startTime = System.nanoTime();

QuickSortStats stats = new QuickSortStats();

quickSortRecursive(orders, 0, orders.length - 1, stats);

long endTime = System.nanoTime();

return new SortingResult("Quick Sort", stats.comparisons, stats.swaps, endTime - startTime);

}

private static void quickSortRecursive(Order[] orders, int low, int high, QuickSortStats stats) {

if (low < high) {

int pivotIndex = partition(orders, low, high, stats);

// Recursively sort elements before and after partition

quickSortRecursive(orders, low, pivotIndex - 1, stats);

quickSortRecursive(orders, pivotIndex + 1, high, stats);

}

}

private static int partition(Order[] orders, int low, int high, QuickSortStats stats) {

// Choose rightmost element as pivot

double pivot = orders[high].getTotalPrice();

int i = low - 1; // Index of smaller element

for (int j = low; j < high; j++) {

stats.comparisons++;

// If current element is greater than or equal to pivot (descending order)

if (orders[j].getTotalPrice() >= pivot) {

i++;

// Swap elements

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

stats.swaps++;

}

}

// Place pivot in correct position

Order temp = orders[i + 1];

orders[i + 1] = orders[high];

orders[high] = temp;

stats.swaps++;

return i + 1;

}

// Insertion Sort - O(n²) time complexity

public static SortingResult insertionSort(Order[] orders) {

long startTime = System.nanoTime();

int comparisons = 0;

int swaps = 0;

int n = orders.length;

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

Order key = orders[i];

int j = i - 1;

// Move elements greater than key one position ahead (descending order)

while (j >= 0) {

comparisons++;

if (orders[j].getTotalPrice() >= key.getTotalPrice()) {

break;

}

orders[j + 1] = orders[j];

swaps++;

j--;

}

orders[j + 1] = key;

if (j + 1 != i) swaps++;

}

long endTime = System.nanoTime();

return new SortingResult("Insertion Sort", comparisons, swaps, endTime - startTime);

}

// Merge Sort - O(n log n) time complexity

public static SortingResult mergeSort(Order[] orders) {

long startTime = System.nanoTime();

MergeSortStats stats = new MergeSortStats();

Order[] tempArray = new Order[orders.length];

mergeSortRecursive(orders, tempArray, 0, orders.length - 1, stats);

long endTime = System.nanoTime();

return new SortingResult("Merge Sort", stats.comparisons, stats.merges, endTime - startTime);

}

private static void mergeSortRecursive(Order[] orders, Order[] temp, int left, int right, MergeSortStats stats) {

if (left < right) {

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

mergeSortRecursive(orders, temp, left, mid, stats);

mergeSortRecursive(orders, temp, mid + 1, right, stats);

merge(orders, temp, left, mid, right, stats);

}

}

private static void merge(Order[] orders, Order[] temp, int left, int mid, int right, MergeSortStats stats) {

int i = left, j = mid + 1, k = left;

// Merge the two halves

while (i <= mid && j <= right) {

stats.comparisons++;

if (orders[i].getTotalPrice() >= orders[j].getTotalPrice()) {

temp[k++] = orders[i++];

} else {

temp[k++] = orders[j++];

}

stats.merges++;

}

// Copy remaining elements

while (i <= mid) {

temp[k++] = orders[i++];

stats.merges++;

}

while (j <= right) {

temp[k++] = orders[j++];

stats.merges++;

}

// Copy back to original array

for (i = left; i <= right; i++) {

orders[i] = temp[i];

}

}

}

// Helper classes for statistics

class QuickSortStats {

int comparisons = 0;

int swaps = 0;

}

class MergeSortStats {

int comparisons = 0;

int merges = 0;

}

// Sorting result wrapper

class SortingResult {

private String algorithm;

private int comparisons;

private int operations; // swaps or merges

private long timeNanoseconds;

public SortingResult(String algorithm, int comparisons, int operations, long timeNanoseconds) {

this.algorithm = algorithm;

this.comparisons = comparisons;

this.operations = operations;

this.timeNanoseconds = timeNanoseconds;

}

public double getTimeMicroseconds() { return timeNanoseconds / 1000.0; }

public double getTimeMilliseconds() { return timeNanoseconds / 1\_000\_000.0; }

@Override

public String toString() {

return String.format("%s: Comparisons=%d, Operations=%d, Time=%.2f μs",

algorithm, comparisons, operations, getTimeMicroseconds());

}

}

// Performance comparison utility

class SortingPerformanceAnalyzer {

public static void compareAllAlgorithms(Order[] originalOrders) {

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

System.out.println("Dataset size: " + originalOrders.length + " orders");

// Test each algorithm

SortingResult[] results = new SortingResult[4];

// Bubble Sort

Order[] bubbleArray = Arrays.copyOf(originalOrders, originalOrders.length);

results[0] = OrderSortingEngine.bubbleSort(bubbleArray);

// Insertion Sort

Order[] insertionArray = Arrays.copyOf(originalOrders, originalOrders.length);

results[1] = OrderSortingEngine.insertionSort(insertionArray);

// Quick Sort

Order[] quickArray = Arrays.copyOf(originalOrders, originalOrders.length);

results[2] = OrderSortingEngine.quickSort(quickArray);

// Merge Sort

Order[] mergeArray = Arrays.copyOf(originalOrders, originalOrders.length);

results[3] = OrderSortingEngine.mergeSort(mergeArray);

// Display results

for (SortingResult result : results) {

System.out.println(result);

}

// Performance analysis

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

double bubbleTime = results[0].getTimeMicroseconds();

double quickTime = results[2].getTimeMicroseconds();

System.out.printf("Quick Sort is %.2fx faster than Bubble Sort\n", bubbleTime / quickTime);

System.out.printf("Quick Sort used %d fewer comparisons than Bubble Sort\n",

results[0].comparisons - results[2].comparisons);

}

public static void demonstrateScalability() {

System.out.println("\n=== SCALABILITY DEMONSTRATION ===");

int[] sizes = {10, 50, 100, 500};

for (int size : sizes) {

System.out.println("\nTesting with " + size + " orders:");

// Generate random orders

Order[] orders = generateRandomOrders(size);

// Test Bubble Sort

Order[] bubbleArray = Arrays.copyOf(orders, orders.length);

SortingResult bubbleResult = OrderSortingEngine.bubbleSort(bubbleArray);

// Test Quick Sort

Order[] quickArray = Arrays.copyOf(orders, orders.length);

SortingResult quickResult = OrderSortingEngine.quickSort(quickArray);

System.out.printf(" Bubble Sort: %.2f ms\n", bubbleResult.getTimeMilliseconds());

System.out.printf(" Quick Sort: %.2f ms\n", quickResult.getTimeMilliseconds());

System.out.printf(" Speedup: %.2fx\n",

bubbleResult.getTimeMicroseconds() / quickResult.getTimeMicroseconds());

}

}

private static Order[] generateRandomOrders(int count) {

Order[] orders = new Order[count];

Random random = new Random();

String[] names = {"Alice", "Bob", "Charlie", "Diana", "Eve", "Frank", "Grace", "Henry"};

String[] statuses = {"Pending", "Processing", "Shipped", "Delivered"};

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

String orderId = "ORD" + String.format("%04d", i + 1);

String customerName = names[random.nextInt(names.length)];

double totalPrice = 10 + random.nextDouble() \* 1990; // $10 to $2000

String status = statuses[random.nextInt(statuses.length)];

orders[i] = new Order(orderId, customerName, totalPrice, status);

}

return orders;

}

}

// Main demonstration class

public class OrderSortingDemo {

public static void main(String[] args) {

// Create sample orders

Order[] orders = {

new Order("ORD001", "Alice Johnson", 1250.99, "Pending"),

new Order("ORD002", "Bob Smith", 89.99, "Processing"),

new Order("ORD003", "Charlie Brown", 2500.00, "Shipped"),

new Order("ORD004", "Diana Prince", 156.75, "Delivered"),

new Order("ORD005", "Eve Wilson", 3200.50, "Pending"),

new Order("ORD006", "Frank Miller", 45.99, "Processing"),

new Order("ORD007", "Grace Lee", 890.25, "Shipped"),

new Order("ORD008", "Henry Davis", 1750.00, "Delivered"),

new Order("ORD009", "Ivy Chen", 299.99, "Pending"),

new Order("ORD010", "Jack Thompson", 125.50, "Processing")

};

// Display original orders

System.out.println("=== ORIGINAL ORDERS ===");

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

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

}

// Demonstrate Bubble Sort

System.out.println("\n=== BUBBLE SORT RESULT ===");

Order[] bubbleSorted = Arrays.copyOf(orders, orders.length);

SortingResult bubbleResult = OrderSortingEngine.bubbleSort(bubbleSorted);

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

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

}

System.out.println(bubbleResult);

// Demonstrate Quick Sort

System.out.println("\n=== QUICK SORT RESULT ===");

Order[] quickSorted = Arrays.copyOf(orders, orders.length);

SortingResult quickResult = OrderSortingEngine.quickSort(quickSorted);

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

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

}

System.out.println(quickResult);

// Performance comparison

SortingPerformanceAnalyzer.compareAllAlgorithms(orders);

// Scalability demonstration

SortingPerformanceAnalyzer.demonstrateScalability();

System.out.println("\n=== RECOMMENDATION ===");

System.out.println("For e-commerce order sorting, Quick Sort is recommended because:");

System.out.println("1. O(n log n) average time complexity vs O(n²) for Bubble Sort");

System.out.println("2. Excellent performance on large datasets");

System.out.println("3. In-place sorting with minimal memory overhead");

System.out.println("4. Handles real-world data efficiently");

}

}

**Exercise 4: Employee Management System**

**Scenario:**

You are developing an employee management system for a company. Efficiently managing employee records is crucial.

Code:

*class Employee {*

*private int employeeId;*

*private String name;*

*private String position;*

*private double salary;*

*public Employee(int employeeId, String name, String position, double salary) {*

*this.employeeId = employeeId;*

*this.name = name;*

*this.position = position;*

*this.salary = salary;*

*}*

*// Getters*

*public int getEmployeeId() { return employeeId; }*

*public String getName() { return name; }*

*public String getPosition() { return position; }*

*public double getSalary() { return salary; }*

*@Override*

*public String toString() {*

*return "Employee{" +*

*"ID=" + employeeId +*

*", name='" + name + '\'' +*

*", position='" + position + '\'' +*

*", salary=" + salary +*

*'}';*

*}*

*}*

*// Employee Management System*

*class EmployeeManagementSystem {*

*private Employee[] employees;*

*private int size;*

*private int capacity;*

*public EmployeeManagementSystem(int capacity) {*

*this.capacity = capacity;*

*this.employees = new Employee[capacity];*

*this.size = 0;*

*}*

*// Add employee - O(1) time complexity*

*public boolean addEmployee(Employee employee) {*

*if (size >= capacity) {*

*System.out.println("Array is full. Cannot add more employees.");*

*return false;*

*}*

*employees[size++] = employee;*

*System.out.println("Employee added successfully.");*

*return true;*

*}*

*// Search employee by ID - O(n) time complexity*

*public Employee searchEmployee(int employeeId) {*

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

*if (employees[i].getEmployeeId() == employeeId) {*

*return employees[i];*

*}*

*}*

*return null;*

*}*

*// Traverse all employees - O(n) time complexity*

*public void traverseEmployees() {*

*if (size == 0) {*

*System.out.println("No employees found.");*

*return;*

*}*

*System.out.println("Employee List:");*

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

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

*}*

*}*

*// Delete employee by ID - O(n) time complexity*

*public boolean deleteEmployee(int employeeId) {*

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

*if (employees[i].getEmployeeId() == employeeId) {*

*// Shift elements to fill the gap*

*for (int j = i; j < size - 1; j++) {*

*employees[j] = employees[j + 1];*

*}*

*employees[--size] = null; // Clear last element*

*System.out.println("Employee deleted successfully.");*

*return true;*

*}*

*}*

*System.out.println("Employee not found.");*

*return false;*

*}*

*public int getSize() {*

*return size;*

*}*

*}*

*// Main class for testing*

*public class Main {*

*public static void main(String[] args) {*

*EmployeeManagementSystem ems = new EmployeeManagementSystem(5);*

*// Add employees*

*ems.addEmployee(new Employee(101, "John Doe", "Developer", 75000));*

*ems.addEmployee(new Employee(102, "Jane Smith", "Manager", 85000));*

*ems.addEmployee(new Employee(103, "Bob Johnson", "Analyst", 65000));*

*// Traverse employees*

*ems.traverseEmployees();*

*// Search employee*

*System.out.println("\nSearching for employee ID 102:");*

*Employee found = ems.searchEmployee(102);*

*if (found != null) {*

*System.out.println("Found: " + found);*

*} else {*

*System.out.println("Employee not found.");*

*}*

*// Delete employee*

*System.out.println("\nDeleting employee ID 102:");*

*ems.deleteEmployee(102);*

*// Traverse after deletion*

*System.out.println("\nAfter deletion:");*

*ems.traverseEmployees();*

*}*

*}*

**Exercise 5: Task Management System**

**Scenario:**

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

Code:

*class Task {*

*private int taskId;*

*private String taskName;*

*private String status;*

*public Task(int taskId, String taskName, String status) {*

*this.taskId = taskId;*

*this.taskName = taskName;*

*this.status = status;*

*}*

*// Getters and Setters*

*public int getTaskId() { return taskId; }*

*public String getTaskName() { return taskName; }*

*public String getStatus() { return status; }*

*public void setStatus(String status) { this.status = status; }*

*@Override*

*public String toString() {*

*return "Task{" +*

*"ID=" + taskId +*

*", name='" + taskName + '\'' +*

*", status='" + status + '\'' +*

*'}';*

*}*

*}*

*// Node class for linked list*

*class TaskNode {*

*Task task;*

*TaskNode next;*

*public TaskNode(Task task) {*

*this.task = task;*

*this.next = null;*

*}*

*}*

*// Singly Linked List Implementation*

*class TaskManagementSystem {*

*private TaskNode head;*

*private int size;*

*public TaskManagementSystem() {*

*this.head = null;*

*this.size = 0;*

*}*

*// Add task at the beginning - O(1) time complexity*

*public void addTask(Task task) {*

*TaskNode newNode = new TaskNode(task);*

*newNode.next = head;*

*head = newNode;*

*size++;*

*System.out.println("Task added successfully: " + task.getTaskName());*

*}*

*// Add task at the end - O(n) time complexity*

*public void addTaskAtEnd(Task task) {*

*TaskNode newNode = new TaskNode(task);*

*if (head == null) {*

*head = newNode;*

*} else {*

*TaskNode current = head;*

*while (current.next != null) {*

*current = current.next;*

*}*

*current.next = newNode;*

*}*

*size++;*

*System.out.println("Task added at end: " + task.getTaskName());*

*}*

*// Search task by ID - O(n) time complexity*

*public Task searchTask(int taskId) {*

*TaskNode current = head;*

*while (current != null) {*

*if (current.task.getTaskId() == taskId) {*

*return current.task;*

*}*

*current = current.next;*

*}*

*return null;*

*}*

*// Traverse all tasks - O(n) time complexity*

*public void traverseTasks() {*

*if (head == null) {*

*System.out.println("No tasks found.");*

*return;*

*}*

*System.out.println("Task List:");*

*TaskNode current = head;*

*int index = 1;*

*while (current != null) {*

*System.out.println(index + ". " + current.task);*

*current = current.next;*

*index++;*

*}*

*}*

*// Delete task by ID - O(n) time complexity*

*public boolean deleteTask(int taskId) {*

*if (head == null) {*

*System.out.println("No tasks to delete.");*

*return false;*

*}*

*// If head node contains the task to delete*

*if (head.task.getTaskId() == taskId) {*

*head = head.next;*

*size--;*

*System.out.println("Task deleted successfully.");*

*return true;*

*}*

*// Search for the task in remaining nodes*

*TaskNode current = head;*

*while (current.next != null) {*

*if (current.next.task.getTaskId() == taskId) {*

*current.next = current.next.next;*

*size--;*

*System.out.println("Task deleted successfully.");*

*return true;*

*}*

*current = current.next;*

*}*

*System.out.println("Task not found.");*

*return false;*

*}*

*// Get size of the list*

*public int getSize() {*

*return size;*

*}*

*// Check if list is empty*

*public boolean isEmpty() {*

*return head == null;*

*}*

*}*

*// Doubly Linked List Node (for comparison)*

*class DoublyTaskNode {*

*Task task;*

*DoublyTaskNode next;*

*DoublyTaskNode prev;*

*public DoublyTaskNode(Task task) {*

*this.task = task;*

*this.next = null;*

*this.prev = null;*

*}*

*}*

*// Main class for testing*

*public class Main {*

*public static void main(String[] args) {*

*TaskManagementSystem tms = new TaskManagementSystem();*

*// Add tasks*

*tms.addTask(new Task(101, "Design Database", "Pending"));*

*tms.addTask(new Task(102, "Implement API", "In Progress"));*

*tms.addTaskAtEnd(new Task(103, "Write Tests", "Pending"));*

*tms.addTask(new Task(104, "Deploy Application", "Not Started"));*

*// Traverse tasks*

*System.out.println("\n=== All Tasks ===");*

*tms.traverseTasks();*

*// Search task*

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

*Task found = tms.searchTask(102);*

*if (found != null) {*

*System.out.println("Found: " + found);*

*} else {*

*System.out.println("Task not found.");*

*}*

*// Delete task*

*System.out.println("\n=== Delete Task ===");*

*tms.deleteTask(102);*

*// Traverse after deletion*

*System.out.println("\n=== After Deletion ===");*

*tms.traverseTasks();*

*System.out.println("\nTotal tasks: " + tms.getSize());*

*}*

*}*

**Exercise 6: Library Management System**

**Scenario:**

You are developing a library management system where users can search for books by title or author.

Code:

*import java.util.Arrays;*

*import java.util.Comparator;*

*// Book class*

*class Book {*

*private int bookId;*

*private String title;*

*private String author;*

*public Book(int bookId, String title, String author) {*

*this.bookId = bookId;*

*this.title = title;*

*this.author = author;*

*}*

*// Getters*

*public int getBookId() { return bookId; }*

*public String getTitle() { return title; }*

*public String getAuthor() { return author; }*

*@Override*

*public String toString() {*

*return "Book{" +*

*"ID=" + bookId +*

*", title='" + title + '\'' +*

*", author='" + author + '\'' +*

*'}';*

*}*

*}*

*// Library Management System*

*class LibraryManagementSystem {*

*private Book[] books;*

*private int size;*

*private int capacity;*

*public LibraryManagementSystem(int capacity) {*

*this.capacity = capacity;*

*this.books = new Book[capacity];*

*this.size = 0;*

*}*

*// Add book to library*

*public boolean addBook(Book book) {*

*if (size >= capacity) {*

*System.out.println("Library is full. Cannot add more books.");*

*return false;*

*}*

*books[size++] = book;*

*return true;*

*}*

*// LINEAR SEARCH - O(n) time complexity*

*public Book linearSearchByTitle(String title) {*

*System.out.println("Performing Linear Search for: " + title);*

*int comparisons = 0;*

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

*comparisons++;*

*if (books[i].getTitle().equalsIgnoreCase(title)) {*

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

*return books[i];*

*}*

*}*

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

*return null;*

*}*

*// LINEAR SEARCH by Author - O(n) time complexity*

*public Book[] linearSearchByAuthor(String author) {*

*System.out.println("Performing Linear Search for author: " + author);*

*Book[] results = new Book[size];*

*int resultCount = 0;*

*int comparisons = 0;*

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

*comparisons++;*

*if (books[i].getAuthor().equalsIgnoreCase(author)) {*

*results[resultCount++] = books[i];*

*}*

*}*

*System.out.println("Found " + resultCount + " books after " + comparisons + " comparisons.");*

*return Arrays.copyOf(results, resultCount);*

*}*

*// Sort books by title for binary search*

*public void sortBooksByTitle() {*

*Arrays.sort(books, 0, size, new Comparator<Book>() {*

*@Override*

*public int compare(Book b1, Book b2) {*

*return b1.getTitle().compareToIgnoreCase(b2.getTitle());*

*}*

*});*

*System.out.println("Books sorted by title for binary search.");*

*}*

*// BINARY SEARCH - O(log n) time complexity*

*// Prerequisite: Array must be sorted by title*

*public Book binarySearchByTitle(String title) {*

*System.out.println("Performing Binary Search for: " + title);*

*int left = 0;*

*int right = size - 1;*

*int comparisons = 0;*

*while (left <= right) {*

*comparisons++;*

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

*int comparison = books[mid].getTitle().compareToIgnoreCase(title);*

*if (comparison == 0) {*

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

*return books[mid];*

*}*

*if (comparison < 0) {*

*left = mid + 1; // Search right half*

*} else {*

*right = mid - 1; // Search left half*

*}*

*}*

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

*return null;*

*}*

*// Recursive Binary Search implementation*

*public Book binarySearchRecursive(String title) {*

*System.out.println("Performing Recursive Binary Search for: " + title);*

*return binarySearchRecursiveHelper(title, 0, size - 1, 0);*

*}*

*private Book binarySearchRecursiveHelper(String title, int left, int right, int comparisons) {*

*if (left > right) {*

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

*return null;*

*}*

*comparisons++;*

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

*int comparison = books[mid].getTitle().compareToIgnoreCase(title);*

*if (comparison == 0) {*

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

*return books[mid];*

*}*

*if (comparison < 0) {*

*return binarySearchRecursiveHelper(title, mid + 1, right, comparisons);*

*} else {*

*return binarySearchRecursiveHelper(title, left, mid - 1, comparisons);*

*}*

*}*

*// Display all books*

*public void displayBooks() {*

*System.out.println("\n=== Library Books ===");*

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

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

*}*

*}*

*public int getSize() {*

*return size;*

*}*

*}*

*// Main class for testing*

*public class Main {*

*public static void main(String[] args) {*

*LibraryManagementSystem library = new LibraryManagementSystem(10);*

*// Add books to library*

*library.addBook(new Book(101, "To Kill a Mockingbird", "Harper Lee"));*

*library.addBook(new Book(102, "1984", "George Orwell"));*

*library.addBook(new Book(103, "Pride and Prejudice", "Jane Austen"));*

*library.addBook(new Book(104, "The Great Gatsby", "F. Scott Fitzgerald"));*

*library.addBook(new Book(105, "Animal Farm", "George Orwell"));*

*library.addBook(new Book(106, "Lord of the Flies", "William Golding"));*

*library.addBook(new Book(107, "Jane Eyre", "Charlotte Bronte"));*

*library.addBook(new Book(108, "Brave New World", "Aldous Huxley"));*

*library.displayBooks();*

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

*System.out.println("LINEAR SEARCH DEMONSTRATIONS");*

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

*// Linear search by title*

*Book found = library.linearSearchByTitle("1984");*

*if (found != null) {*

*System.out.println("Result: " + found);*

*}*

*System.out.println();*

*// Linear search by author*

*Book[] authorBooks = library.linearSearchByAuthor("George Orwell");*

*System.out.println("Books by George Orwell:");*

*for (Book book : authorBooks) {*

*System.out.println("- " + book);*

*}*

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

*System.out.println("BINARY SEARCH DEMONSTRATIONS");*

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

*// Sort books for binary search*

*library.sortBooksByTitle();*

*library.displayBooks();*

*System.out.println();*

*// Binary search by title*

*Book binaryFound = library.binarySearchByTitle("Pride and Prejudice");*

*if (binaryFound != null) {*

*System.out.println("Result: " + binaryFound);*

*}*

*System.out.println();*

*// Recursive binary search*

*Book recursiveFound = library.binarySearchRecursive("Animal Farm");*

*if (recursiveFound != null) {*

*System.out.println("Result: " + recursiveFound);*

*}*

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

*System.out.println("PERFORMANCE COMPARISON");*

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

*// Compare search performance*

*System.out.println("Searching for 'Brave New World':");*

*library.linearSearchByTitle("Brave New World");*

*library.binarySearchByTitle("Brave New World");*

*}*

*}*

**Exercise 7: Financial Forecasting**

**Scenario:**

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

Code:

Code:

*import java.util.HashMap;*

*import java.util.Map;*

*// Financial Data Point class*

*class FinancialData {*

*private int period;*

*private double value;*

*private double growthRate;*

*public FinancialData(int period, double value, double growthRate) {*

*this.period = period;*

*this.value = value;*

*this.growthRate = growthRate;*

*}*

*public int getPeriod() { return period; }*

*public double getValue() { return value; }*

*public double getGrowthRate() { return growthRate; }*

*@Override*

*public String toString() {*

*return String.format("Period %d: $%.2f (Growth: %.2f%%)",*

*period, value, growthRate \* 100);*

*}*

*}*

*// Financial Forecasting System*

*class FinancialForecastingSystem {*

*private static int recursiveCallCount = 0;*

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

*// Basic Recursive Future Value Calculator*

*// Formula: FV = PV \* (1 + r)^n*

*public static double calculateFutureValueRecursive(double presentValue,*

*double growthRate, int periods) {*

*recursiveCallCount++;*

*// Base case*

*if (periods == 0) {*

*return presentValue;*

*}*

*// Recursive case: FV(n) = FV(n-1) \* (1 + growthRate)*

*return calculateFutureValueRecursive(presentValue, growthRate, periods - 1)*

*\* (1 + growthRate);*

*}*

*// Optimized Recursive with Memoization*

*public static double calculateFutureValueMemoized(double presentValue,*

*double growthRate, int periods) {*

*String key = presentValue + "," + growthRate + "," + periods;*

*if (memoCache.containsKey(key)) {*

*return memoCache.get(key);*

*}*

*recursiveCallCount++;*

*// Base case*

*if (periods == 0) {*

*memoCache.put(key, presentValue);*

*return presentValue;*

*}*

*// Recursive case with memoization*

*double result = calculateFutureValueMemoized(presentValue, growthRate, periods - 1)*

*\* (1 + growthRate);*

*memoCache.put(key, result);*

*return result;*

*}*

*// Compound Annual Growth Rate (CAGR) Calculator - Recursive*

*public static double calculateCAGRRecursive(double initialValue, double finalValue,*

*int periods) {*

*recursiveCallCount++;*

*// Base case*

*if (periods == 1) {*

*return (finalValue / initialValue) - 1;*

*}*

*// Recursive approach: CAGR = (finalValue/initialValue)^(1/periods) - 1*

*// Using recursive power calculation*

*double ratio = finalValue / initialValue;*

*return Math.pow(ratio, 1.0 / periods) - 1;*

*}*

*// Fibonacci-based Growth Model (Economic cycles)*

*public static double fibonacciGrowthModel(int period) {*

*recursiveCallCount++;*

*// Base cases*

*if (period <= 1) {*

*return period;*

*}*

*// Recursive case: F(n) = F(n-1) + F(n-2)*

*return fibonacciGrowthModel(period - 1) + fibonacciGrowthModel(period - 2);*

*}*

*// Optimized Fibonacci with Memoization*

*private static Map<Integer, Double> fibCache = new HashMap<>();*

*public static double fibonacciGrowthModelOptimized(int period) {*

*if (fibCache.containsKey(period)) {*

*return fibCache.get(period);*

*}*

*recursiveCallCount++;*

*// Base cases*

*if (period <= 1) {*

*fibCache.put(period, (double) period);*

*return period;*

*}*

*// Recursive case with memoization*

*double result = fibonacciGrowthModelOptimized(period - 1) +*

*fibonacciGrowthModelOptimized(period - 2);*

*fibCache.put(period, result);*

*return result;*

*}*

*// Present Value Calculator - Recursive*

*// PV = FV / (1 + r)^n*

*public static double calculatePresentValueRecursive(double futureValue,*

*double discountRate, int periods) {*

*recursiveCallCount++;*

*// Base case*

*if (periods == 0) {*

*return futureValue;*

*}*

*// Recursive case: PV(n) = PV(n-1) / (1 + discountRate)*

*return calculatePresentValueRecursive(futureValue, discountRate, periods - 1)*

*/ (1 + discountRate);*

*}*

*// Net Present Value (NPV) Calculator - Recursive*

*public static double calculateNPVRecursive(double[] cashFlows, double discountRate,*

*int period) {*

*recursiveCallCount++;*

*// Base case*

*if (period >= cashFlows.length) {*

*return 0;*

*}*

*// Recursive case: NPV = CF(0) + CF(1)/(1+r) + CF(2)/(1+r)^2 + ...*

*double presentValueOfCurrentCashFlow = cashFlows[period] /*

*Math.pow(1 + discountRate, period);*

*return presentValueOfCurrentCashFlow +*

*calculateNPVRecursive(cashFlows, discountRate, period + 1);*

*}*

*// Tail-Recursive Optimization for Future Value*

*public static double calculateFutureValueTailRecursive(double presentValue,*

*double growthRate, int periods) {*

*return calculateFutureValueTailHelper(presentValue, growthRate, periods, 1.0);*

*}*

*private static double calculateFutureValueTailHelper(double presentValue,*

*double growthRate, int periods,*

*double accumulator) {*

*recursiveCallCount++;*

*// Base case*

*if (periods == 0) {*

*return presentValue \* accumulator;*

*}*

*// Tail-recursive call*

*return calculateFutureValueTailHelper(presentValue, growthRate, periods - 1,*

*accumulator \* (1 + growthRate));*

*}*

*// Reset counters and cache*

*public static void resetCounters() {*

*recursiveCallCount = 0;*

*memoCache.clear();*

*fibCache.clear();*

*}*

*public static int getRecursiveCallCount() {*

*return recursiveCallCount;*

*}*

*}*

*// Main class for testing and demonstration*

*public class Main {*

*public static void main(String[] args) {*

*System.out.println("=".repeat(60));*

*System.out.println("FINANCIAL FORECASTING SYSTEM - RECURSIVE ALGORITHMS");*

*System.out.println("=".repeat(60));*

*// Test data*

*double initialInvestment = 10000.0;*

*double growthRate = 0.08; // 8% annual growth*

*int periods = 10;*

*System.out.println("Initial Investment: $" + initialInvestment);*

*System.out.println("Growth Rate: " + (growthRate \* 100) + "%");*

*System.out.println("Periods: " + periods + " years");*

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

*System.out.println("1. BASIC RECURSIVE FUTURE VALUE CALCULATION");*

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

*FinancialForecastingSystem.resetCounters();*

*double futureValue = FinancialForecastingSystem.calculateFutureValueRecursive(*

*initialInvestment, growthRate, periods);*

*System.out.printf("Future Value: $%.2f%n", futureValue);*

*System.out.println("Recursive calls made: " +*

*FinancialForecastingSystem.getRecursiveCallCount());*

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

*System.out.println("2. MEMOIZED RECURSIVE CALCULATION");*

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

*FinancialForecastingSystem.resetCounters();*

*double memoizedFV = FinancialForecastingSystem.calculateFutureValueMemoized(*

*initialInvestment, growthRate, periods);*

*System.out.printf("Future Value (Memoized): $%.2f%n", memoizedFV);*

*System.out.println("Recursive calls made: " +*

*FinancialForecastingSystem.getRecursiveCallCount());*

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

*System.out.println("3. COMPOUND ANNUAL GROWTH RATE (CAGR)");*

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

*FinancialForecastingSystem.resetCounters();*

*double cagr = FinancialForecastingSystem.calculateCAGRRecursive(*

*initialInvestment, futureValue, periods);*

*System.out.printf("Calculated CAGR: %.4f (%.2f%%)%n", cagr, cagr \* 100);*

*System.out.println("Recursive calls made: " +*

*FinancialForecastingSystem.getRecursiveCallCount());*

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

*System.out.println("4. FIBONACCI GROWTH MODEL COMPARISON");*

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

*int fibPeriod = 15;*

*// Basic Fibonacci (inefficient)*

*FinancialForecastingSystem.resetCounters();*

*long startTime = System.nanoTime();*

*double fibBasic = FinancialForecastingSystem.fibonacciGrowthModel(fibPeriod);*

*long basicTime = System.nanoTime() - startTime;*

*int basicCalls = FinancialForecastingSystem.getRecursiveCallCount();*

*// Optimized Fibonacci (with memoization)*

*FinancialForecastingSystem.resetCounters();*

*startTime = System.nanoTime();*

*double fibOptimized = FinancialForecastingSystem.fibonacciGrowthModelOptimized(fibPeriod);*

*long optimizedTime = System.nanoTime() - startTime;*

*int optimizedCalls = FinancialForecastingSystem.getRecursiveCallCount();*

*System.out.println("Fibonacci Period: " + fibPeriod);*

*System.out.printf("Basic Result: %.0f (Calls: %d, Time: %.2f ms)%n",*

*fibBasic, basicCalls, basicTime / 1\_000\_000.0);*

*System.out.printf("Optimized Result: %.0f (Calls: %d, Time: %.2f ms)%n",*

*fibOptimized, optimizedCalls, optimizedTime / 1\_000\_000.0);*

*System.out.printf("Performance Improvement: %.2fx faster%n",*

*(double) basicTime / optimizedTime);*

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

*System.out.println("5. NET PRESENT VALUE (NPV) CALCULATION");*

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

*double[] cashFlows = {-10000, 2000, 3000, 4000, 5000, 6000}; // Initial investment + returns*

*double discountRate = 0.10; // 10% discount rate*

*FinancialForecastingSystem.resetCounters();*

*double npv = FinancialForecastingSystem.calculateNPVRecursive(*

*cashFlows, discountRate, 0);*

*System.out.println("Cash Flows: ");*

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

*System.out.printf("Year %d: $%.2f%n", i, cashFlows[i]);*

*}*

*System.out.printf("NPV at %.1f%% discount rate: $%.2f%n",*

*discountRate \* 100, npv);*

*System.out.println("Recursive calls made: " +*

*FinancialForecastingSystem.getRecursiveCallCount());*

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

*System.out.println("6. TAIL-RECURSIVE OPTIMIZATION");*

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

*FinancialForecastingSystem.resetCounters();*

*double tailRecursiveFV = FinancialForecastingSystem.calculateFutureValueTailRecursive(*

*initialInvestment, growthRate, periods);*

*System.out.printf("Future Value (Tail-Recursive): $%.2f%n", tailRecursiveFV);*

*System.out.println("Recursive calls made: " +*

*FinancialForecastingSystem.getRecursiveCallCount());*

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

*System.out.println("SUMMARY AND RECOMMENDATIONS");*

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

*System.out.println("✓ All recursive methods produced consistent results");*

*System.out.println("✓ Memoization dramatically reduces recursive calls");*

*System.out.println("✓ Tail recursion is more memory efficient");*

*System.out.println("✓ Consider iterative solutions for production systems");*

*}*

*}*