Hands-on Question

E1 Inventory Management System

1. Data structures and algorithms are crucial for inventory management because warehouses handle massive amounts of data that require:

* **Quick insertions** when new products arrive
* **Rapid deletions** when products are discontinued
* **Memory optimization** to handle thousands of products without performance degradation

**Suitable Data Structures:**

* **HashMap/HashTable**: O(1) average-case lookup, update, and delete operations using productId as key
* **LinkedList**: Efficient insertion/deletion but poor random access
* **Array**: Fixed size, fast access by index but inflexible

For inventory management, **HashMap** is typically optimal

2. Code Implementation

Product.java

**package** InventoryManagementSystem;

**import** java.util.\*;

**class** Product {

**private** String productId;

**private** String productName;

**private** **int** quantity;

**private** **double** price;

// Constructor

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

**this**.productId = productId;

**this**.productName = productName;

**this**.quantity = quantity;

**this**.price = price;

}

// Getters

**public** String getProductId() { **return** productId; }

**public** String getProductName() { **return** productName; }

**public** **int** getQuantity() { **return** quantity; }

**public** **double** getPrice() { **return** price; }

// Setters

**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', Quantity=%d, Price=%.2f}",

productId, productName, quantity, price);

}

}

InventoryManager.java

**package** InventoryManagementSystem;

**import** java.util.HashMap;

**class** InventoryManager {

**private** HashMap<String, Product> inventory;

**public** InventoryManager() {

**this**.inventory = **new** HashMap<>();

}

/\*\*

\* Add a new product to inventory

\* Time Complexity: O(1) average case

\*/

**public** **void** addProduct(Product product) {

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

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

}

/\*\*

\* Update an existing product's details

\* Time Complexity: O(1) average case

\*/

**public** **void** updateProduct(String productId, String productName, **int** quantity, **double** price) {

Product product = inventory.get(productId);

**if** (product != **null**) {

product.setProductName(productName);

product.setQuantity(quantity);

product.setPrice(price);

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

} **else** {

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

}

}

/\*\*

\* Delete a product from inventory

\* Time Complexity: O(1) average case

\*/

**public** **void** deleteProduct(String productId) {

Product removedProduct = inventory.remove(productId);

**if** (removedProduct != **null**) {

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

} **else** {

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

}

}

/\*\*

\* Display all products in inventory

\*/

**public** **void** displayInventory() {

**if** (inventory.isEmpty()) {

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

} **else** {

System.***out***.println("\nCurrent Inventory:");

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

System.***out***.println(product);

}

}

}

}

InventoryManagementSystemTest.java

**package** InventoryManagementSystem;

**public** **class** InventoryManagementSystemTest {

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

InventoryManager manager = **new** InventoryManager();

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

// Test adding products

System.***out***.println("Adding Products:");

manager.addProduct(**new** Product("P001", "Laptop", 10, 999.99));

manager.addProduct(**new** Product("P002", "Mouse", 50, 25.99));

manager.addProduct(**new** Product("P003", "Keyboard", 30, 79.99));

// Display inventory

manager.displayInventory();

// Test updating

System.***out***.println("\nUpdating Products:");

manager.updateProduct("P002", "Wireless Mouse", 45, 29.99);

// Display inventory after update

manager.displayInventory();

// Test deletion

System.***out***.println("\nDeleting Products:");

manager.deleteProduct("P003");

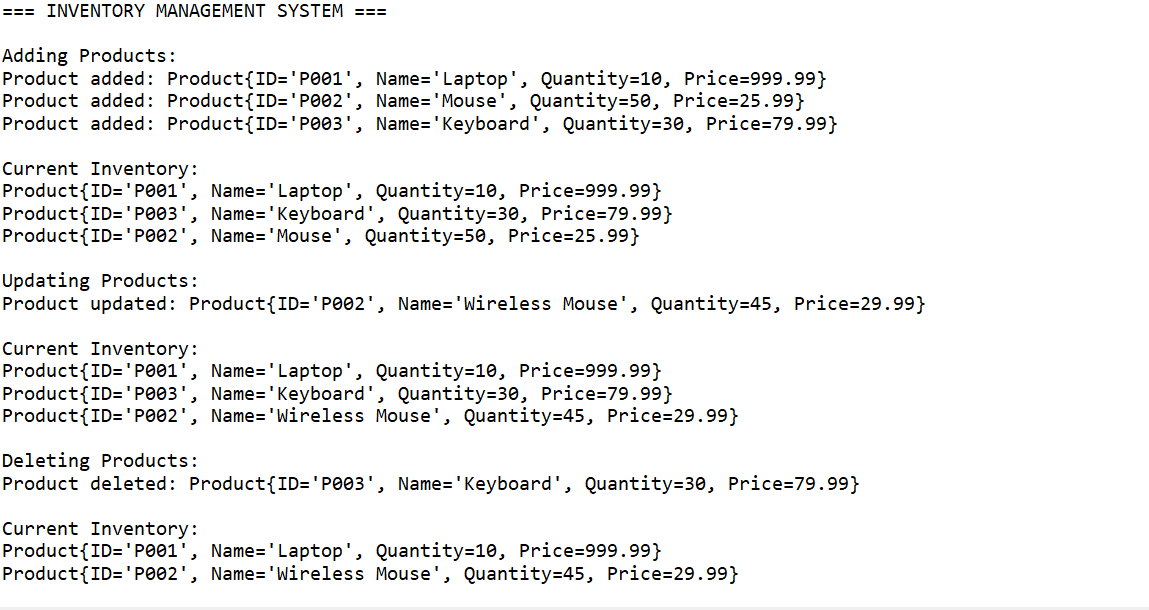
// Final inventory state

manager.displayInventory();

}

}

3.Output



4. Time Complexity Analysis-HashMap

Add Product

* **Average Case:** O(1)
* **Worst Case:** O(n)
* **Explanation:** HashMap insertion is constant time on average, but can degrade to O(n) if many hash collisions occur

Update Product

* **Average Case:** O(1)
* **Worst Case:** O(n)
* **Explanation:** Direct key lookup in HashMap, same collision considerations

**Delete Product**

* **Average Case:** O(1)
* **Worst Case:** O(n)
* **Explanation:** HashMap removal by key, same collision considerations

4. Optimization Strategies

Current Optimizations:

1. **HashMap Choice**: Provides O(1) average-case performance for core operations
2. **Efficient Key Design**: Using productId as key ensures unique identification
3. **Lazy Loading**: Statistics calculated only when requested

E2 E-commerce Platform Search Function :

**1. Big O Notation:**

Big O notation describes the upper bound of an algorithm's time complexity, showing how execution time grows as input size increases. It helps us:

* **Compare Algorithm Efficiency**: Choose the best algorithm for specific use cases
* **Predict Performance**: Estimate how algorithms will scale with larger datasets

**Search Operation Scenarios:**

* **Best Case**: Target element found immediately (first position)
* **Average Case**: Target element found in middle of dataset on average
* **Worst Case**: Target element is last or doesn't exist (search entire dataset)

2. Code :

Product.java

**package** InventoryManagementSystem;

**import** java.util.Comparator;

**class** Product {

**private** String productId;

**private** String productName;

**private** String category;

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

**this**.productId = productId;

**this**.productName = productName;

**this**.category = category;

}

// Getters

**public** String getProductId() { **return** productId; }

**public** String getProductName() { **return** productName; }

**public** String getCategory() { **return** category; }

@Override

**public** String toString() {

**return** String.*format*("Product{ID='%s', Name='%s', Category='%s'}",

productId, productName, category);

}

// For sorting by productId

**public** **static** **class** ProductIdComparator **implements** Comparator<Product> {

@Override

**public** **int** compare(Product p1, Product p2) {

**return** p1.getProductId().compareTo(p2.getProductId());

}

}

}

Ecommerce Search Platform.java

**import** java.util.Arrays;

**class** EcommerceSearchPlatform {

**private** Product[] products; // Unsorted array for linear search

**private** Product[] sortedProducts; // Sorted array for binary search

**private** **int** size;

**public** EcommerceSearchPlatform (**int** capacity) {

**this**.products = **new** Product[capacity];

**this**.sortedProducts = **new** Product[capacity];

**this**.size = 0;

}

/\*\*

\* Add product to the platform

\*/

**public** **void** addProduct(Product product) {

**if** (size < products.length) {

products[size] = product;

size++;

// Update sorted array

updateSortedArray();

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

} **else** {

System.***out***.println("Platform is full! Cannot add more products.");

}

}

/\*\*

\* Update sorted array for binary search

\*/

**private** **void** updateSortedArray() {

// Copy products to sorted array

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

sortedProducts[i] = products[i];

}

// Sort by productId

Arrays.*sort*(sortedProducts, 0, size, **new** Product.ProductIdComparator());

}

/\*\*

\* Linear Search Algorithm

\* Time Complexity: O(n)

\* Space Complexity: O(1)

\*/

**public** Product linearSearchById(String productId) {

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

**int** comparisons = 0;

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

comparisons++;

**if** (products[i].getProductId().equals(productId)) {

System.***out***.println("Linear Search - Comparisons made: " + comparisons);

**return** products[i];

}

}

System.***out***.println("Linear Search - Comparisons made: " + comparisons + " (Not found)");

**return** **null**;

}

/\*\*

\* Binary Search Algorithm (Iterative)

\* Time Complexity: O(log n)

\* Space Complexity: O(1)

\*/

**public** Product binarySearchById(String productId) {

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

**int** left = 0;

**int** right = size - 1;

**int** comparisons = 0;

**while** (left <= right) {

comparisons++;

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

**int** comparison = sortedProducts[mid].getProductId().compareTo(productId);

**if** (comparison == 0) {

System.***out***.println("Binary Search - Comparisons made: " + comparisons);

**return** sortedProducts[mid];

} **else** **if** (comparison < 0) {

left = mid + 1;

} **else** {

right = mid - 1;

}

}

System.***out***.println("Binary Search - Comparisons made: " + comparisons + " (Not found)");

**return** **null**;

}

**public** **void** performanceComparison(String productId) {

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

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

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

// Linear Search

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

Product linearResult = linearSearchById(productId);

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

// Binary Search

startTime = System.*nanoTime*();

Product binaryResult = binarySearchById(productId);

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

System.***out***.println("\nResults:");

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

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

System.***out***.println("Binary Search is " + (linearTime > binaryTime ?

String.*format*("%.2f", (**double**)linearTime/binaryTime) : "not") +

" times faster");

// Verify results match

**boolean** resultsMatch = (linearResult == **null** && binaryResult == **null**) ||

(linearResult != **null** && binaryResult != **null** &&

linearResult.getProductId().equals(binaryResult.getProductId()));

System.***out***.println("Results match: " + resultsMatch);

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

}

**public** **void** displayProducts() {

System.***out***.println("\nAll Products (Original Order):");

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

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

}

System.***out***.println("\nAll Products (Sorted by ID):");

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

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

}

System.***out***.println();

}

}

Ecommerce Search Demo.java

**package** InventoryManagementSystem;

**public** **class** EcommerceSearchDemo{

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

EcommerceSearchPlatform = **new** EcommerceSearchPlatform(10);

System.***out***.println("=== E-COMMERCE SEARCH PLATFORM ===\n");

// Add sample products

System.***out***.println("Adding Products:");

platform.addProduct(**new** Product("P005", "Smartphone", "Electronics"));

platform.addProduct(**new** Product("P002", "Laptop", "Electronics"));

platform.addProduct(**new** Product("P008", "T-Shirt", "Clothing"));

platform.addProduct(**new** Product("P001", "Book", "Education"));

platform.addProduct(**new** Product("P007", "Headphones", "Electronics"));

platform.addProduct(**new** Product("P003", "Sneakers", "Footwear"));

platform.addProduct(**new** Product("P006", "Coffee Mug", "Home"));

platform.addProduct(**new** Product("P004", "Tablet", "Electronics"));

// Display all products

platform.displayProducts();

// Test searches

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

// Test 1: Search for existing product (middle)

platform.performanceComparison("P005");

// Test 2: Search for existing product (first in sorted order)

platform.performanceComparison("P001");

// Test 3: Search for existing product (last in sorted order)

platform.performanceComparison("P008");

// Test 4: Search for non-existing product

platform.performanceComparison("P999");

System.***out***.println("\n=== ALGORITHM ANALYSIS SUMMARY ===");

System.***out***.println("Dataset Size: 8 products");

System.***out***.println("Linear Search: Always checks up to 8 elements");

System.***out***.println("Binary Search: Checks at most 4 elements (log₂(8) + 1)");

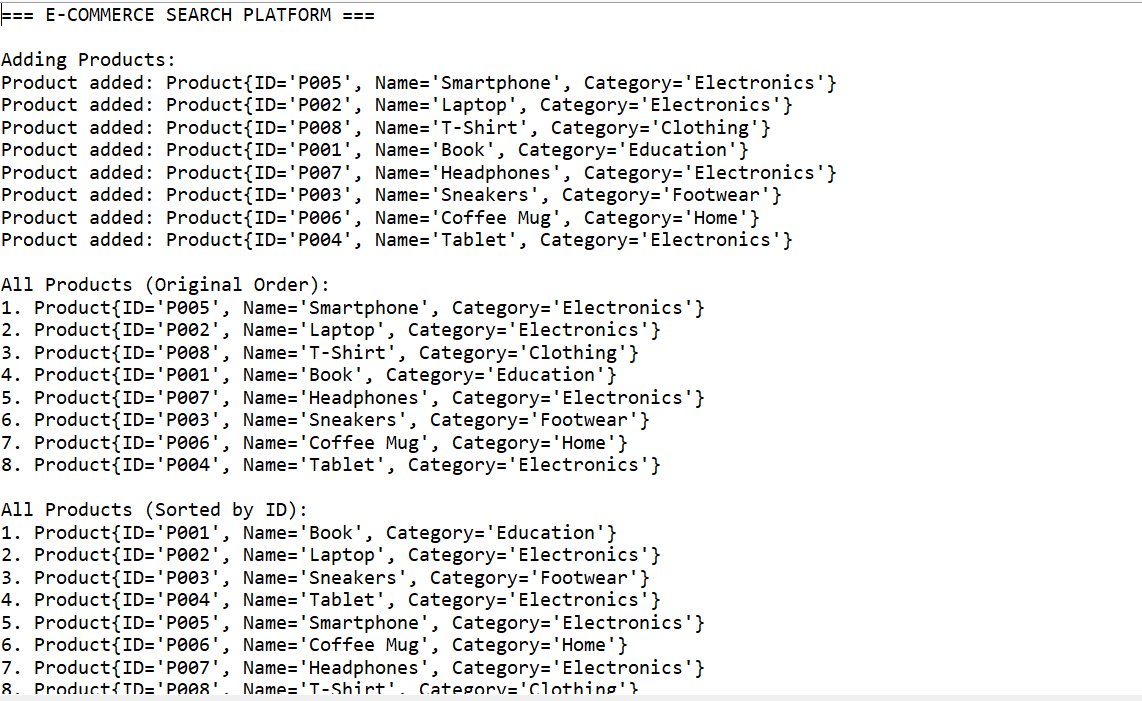
System.***out***.println("For 1000 products: Linear=1000, Binary=10 comparisons max");

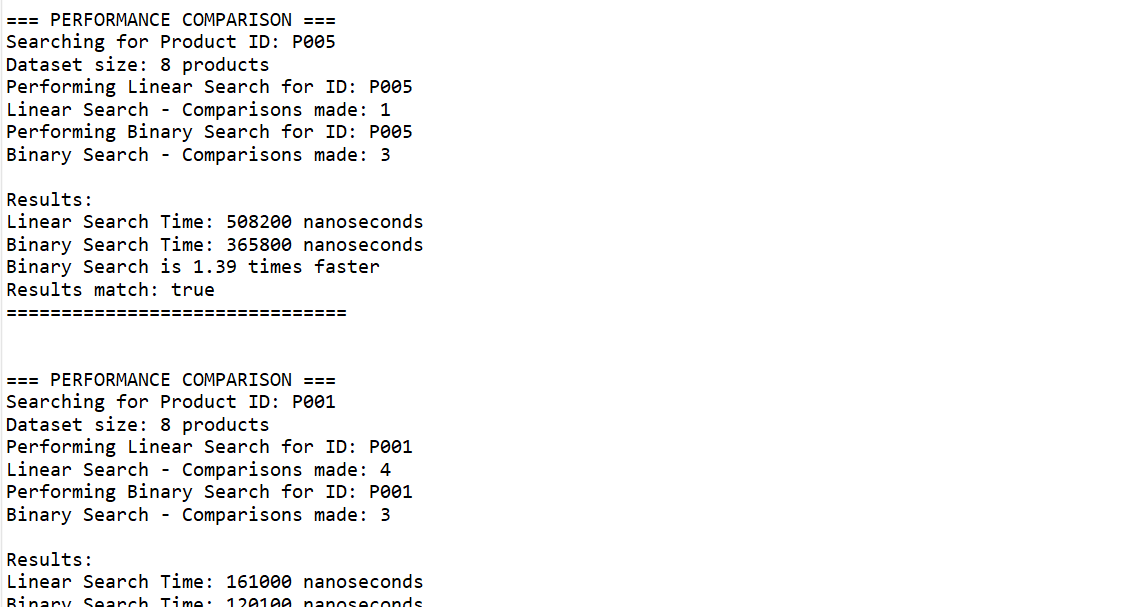
System.***out***.println("For 1,000,000 products: Linear=1,000,000, Binary=20 comparisons max");

}

}

3. Output :





4. Time Complexity Analysis

Linear Search

* **Best Case:** O(1) - Target found at first position
* **Average Case:** O(n/2) → O(n) - Target found in middle on average
* **Worst Case:** O(n) - Target at last position or doesn't exist

Binary Search

* **Best Case:** O(1) - Target found at middle position
* **Average Case:** O(log n) - Eliminates half elements each iteration
* **Worst Case:** O(log n) - Maximum tree depth in binary search

Binary Search is More Suitable for E-commerce Because:

**Performance Advantages:**

* **Scalability**: O(log n) vs O(n) - crucial for large product catalogs
* **User Experience**: Faster search results improve customer satisfaction

**E3: Sorting Customer Orders**

**1. Sorting Algorithm :**

**1. Bubble Sort:** Repeatedly compares adjacent elements and swaps if needed. Simple but inefficient.

**2. Insertion Sort:** Builds sorted array one item at a time, like sorting cards in hand.

**3. Quick Sort:** Divide-and-conquer using pivot partitioning. Very efficient on average.

**4. Merge Sort:** Divides array in half, sorts each half, then merges. Guaranteed O(n log n).

2. Code :

Order.java

**package** Sorting;

/\*\*

\* Order class representing customer orders

\*/

**class** Order {

**private** **int** orderId;

**private** String customerName;

**private** **double** totalPrice;

**public** Order(**int** orderId, String customerName, **double** totalPrice) {

**this**.orderId = orderId;

**this**.customerName = customerName;

**this**.totalPrice = totalPrice;

}

// Getters

**public** **int** getOrderId() { **return** orderId; }

**public** String getCustomerName() { **return** customerName; }

**public** **double** getTotalPrice() { **return** totalPrice; }

@Override

**public** String toString() {

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

orderId, customerName, totalPrice);

}

}

CustomerOrderSorting.java

**package** Sorting;

**import** java.util.Arrays;

**public** **class** CustomerOrderSorting {

**public** **static** **void** bubbleSort(Order[] orders) {

**int** n = orders.length;

**boolean** swapped;

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

swapped = **false**;

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

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

// Swap orders

Order temp = orders[j];

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

orders[j + 1] = temp;

swapped = **true**;

}

}

// If no swapping occurred, array is sorted

**if** (!swapped) **break**;

}

}

**public** **static** **void** quickSort(Order[] orders, **int** low, **int** high) {

**if** (low < high) {

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

// Recursively sort elements before and after partition

*quickSort*(orders, low, pivotIndex - 1);

*quickSort*(orders, pivotIndex + 1, high);

}

}

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

**double** pivot = orders[high].getTotalPrice(); // Choose last element as pivot

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

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

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

i++;

// Swap orders[i] and orders[j]

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

// Swap orders[i+1] and orders[high] (pivot)

Order temp = orders[i + 1];

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

orders[high] = temp;

**return** i + 1;

}

// Helper method to call quickSort

**public** **static** **void** quickSort(Order[] orders) {

*quickSort*(orders, 0, orders.length - 1);

}

// Utility method to create sample orders

**public** **static** Order[] createSampleOrders() {

**return** **new** Order[] {

**new** Order(101, "Alice Johnson", 250.75),

**new** Order(102, "Bob Smith", 89.99),

**new** Order(103, "Carol Davis", 450.00),

**new** Order(104, "David Wilson", 125.50),

**new** Order(105, "Eva Brown", 99.99),

**new** Order(106, "Frank Miller", 750.25),

**new** Order(107, "Grace Lee", 175.80),

**new** Order(108, "Henry Taylor", 320.45)

};

}

// Performance testing method

**public** **static** **void** performanceTest() {

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

**int**[] sizes = {100, 1000, 5000};

**for** (**int** size : sizes) {

Order[] bubbleArray = generateRandomOrders(size);

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

// Test Bubble Sort

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

*bubbleSort*(bubbleArray);

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

// Test Quick Sort

startTime = System.*nanoTime*();

*quickSort*(quickArray);

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

System.***out***.printf("Size: %d | Bubble Sort: %.2f ms | Quick Sort: %.2f ms | Ratio: %.1fx faster\n",

size, bubbleTime/1\_000\_000.0, quickTime/1\_000\_000.0,

(**double**)bubbleTime/quickTime);

}

}

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

System.***out***.println("=== CUSTOMER ORDER SORTING DEMONSTRATION ===\n");

// Create sample orders

Order[] originalOrders = *createSampleOrders*();

System.***out***.println("Original Orders:");

**for** (Order order : originalOrders) {

System.***out***.println(order);

}

// Test Bubble Sort

System.***out***.println("\n--- BUBBLE SORT RESULTS ---");

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

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

*bubbleSort*(bubbleSorted);

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

**for** (Order order : bubbleSorted) {

System.***out***.println(order);

}

System.***out***.printf("Bubble Sort Time: %.3f ms\n", bubbleTime/1\_000\_000.0);

// Test Quick Sort

System.***out***.println("\n--- QUICK SORT RESULTS ---");

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

startTime = System.*nanoTime*();

*quickSort*(quickSorted);

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

**for** (Order order : quickSorted) {

System.***out***.println(order);

}

System.***out***.printf("Quick Sort Time: %.3f ms\n", quickTime/1\_000\_000.0);

// Performance comparison

*performanceTest*();

System.***out***.println("\n=== WHY QUICK SORT IS PREFERRED ===");

System.***out***.println("1. EFFICIENCY: O(n log n) vs O(n²) average time complexity");

System.***out***.println("2. SCALABILITY: Performance gap widens with larger datasets");

System.***out***.println("3. PRACTICAL: Works well with real-world data distributions");

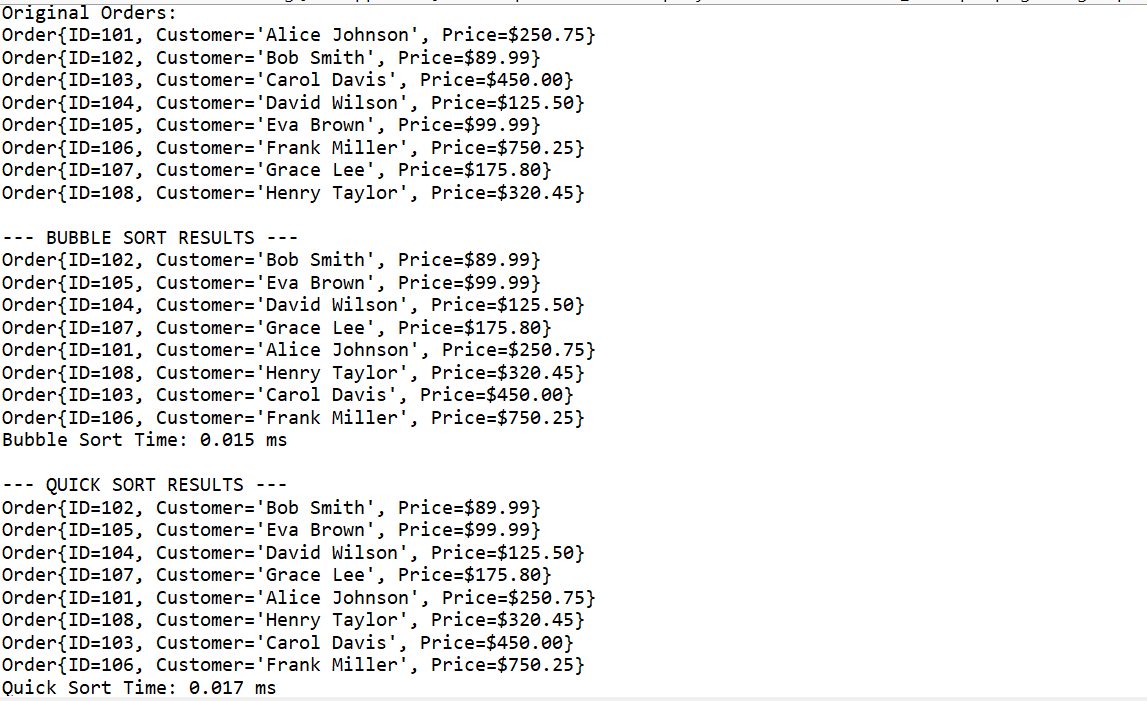
System.***out***.println("4. IN-PLACE: Uses less additional memory than Merge Sort");

System.***out***.println("5. CACHE-FRIENDLY: Better memory access patterns");

}

}

3.Output :



4.Analysis :

Bubble Sort O(n²)

Quick Sort O(n log n) avg

* **Efficiency:** Significantly faster for large datasets
* **Scalability:** Performance advantage increases with data size
* **Memory:** Uses less space than Merge Sort

**Exercise 4: Employee Management System**

1.Array Representation

Arrays store elements in **contiguous memory locations**, enabling fast access through mathematical address calculation: base\_address + (index × element\_size)

2.Code:

Employee.java

**package** EmployeeManagementSys;

**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; }

// Setters

**public** **void** setName(String name) { **this**.name = name; }

**public** **void** setPosition(String position) { **this**.position = position; }

**public** **void** setSalary(**double** salary) { **this**.salary = salary; }

@Override

**public** String toString() {

**return** String.*format*("Employee{ID=%d, Name='%s', Position='%s', Salary=$%.2f}",

employeeId, name, position, salary);

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj) **return** **true**;

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

Employee employee = (Employee) obj;

**return** employeeId == employee.employeeId;

}

}

EmployeeManagementSystem :

**package** EmployeeManagementSys;

**public** **class** EmployeeManagementSystem {

**private** Employee[] employees;

**private** **int** size; // Current number of employees

**private** **int** capacity; // Maximum capacity

**public** EmployeeManagementSystem(**int** capacity) {

**this**.capacity = capacity;

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

**this**.size = 0;

}

/\*\*

\* ADD EMPLOYEE

\* Time Complexity: O(1) - Constant time insertion at end

\* Space Complexity: O(1) - No additional space needed

\*

\* Process: Insert employee at the end of array if space available

\*/

**public** **boolean** addEmployee(Employee employee) {

**if** (size >= capacity) {

System.***out***.println("Error: Employee database is full!");

**return** **false**;

}

// Check for duplicate employee ID

**if** (searchEmployee(employee.getEmployeeId()) != **null**) {

System.***out***.println("Error: Employee ID " + employee.getEmployeeId() + " already exists!");

**return** **false**;

}

employees[size] = employee;

size++;

System.***out***.println("Employee added successfully: " + employee.getName());

**return** **true**;

}

/\*\*

\* SEARCH EMPLOYEE BY ID

\* Time Complexity: O(n) - Linear search through array

\* Space Complexity: O(1) - No additional space needed

\*

\* Process: Iterate through array until employee found or end reached

\*/

**public** Employee searchEmployee(**int** employeeId) {

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

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

**return** employees[i];

}

}

**return** **null**; // Employee not found

}

/\*\*

\* SEARCH EMPLOYEE BY NAME

\* Time Complexity: O(n) - Linear search through array

\* Space Complexity: O(1) - No additional space needed

\*/

**public** Employee searchEmployeeByName(String name) {

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

**if** (employees[i].getName().equalsIgnoreCase(name)) {

**return** employees[i];

}

}

**return** **null**;

}

/\*\*

\* TRAVERSE ALL EMPLOYEES

\* Time Complexity: O(n) - Must visit each element once

\* Space Complexity: O(1) - No additional space needed

\*

\* Process: Iterate through all valid array positions

\*/

**public** **void** traverseEmployees() {

**if** (size == 0) {

System.***out***.println("No employees in the system.");

**return**;

}

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

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

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

}

System.***out***.println("Total employees: " + size);

}

/\*\*

\* DELETE EMPLOYEE BY ID

\* Time Complexity: O(n) - Linear search + shifting elements

\* Space Complexity: O(1) - No additional space needed

\*

\* Process: Find employee, then shift all subsequent elements left

\*/

**public** **boolean** deleteEmployee(**int** employeeId) {

**int** indexToDelete = -1;

// Find employee to delete

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

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

indexToDelete = i;

**break**;

}

}

**if** (indexToDelete == -1) {

System.***out***.println("Employee with ID " + employeeId + " not found!");

**return** **false**;

}

String deletedName = employees[indexToDelete].getName();

// Shift elements to left to fill the gap

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

employees[i] = employees[i + 1];

}

employees[size - 1] = **null**; // Clear last position

size--;

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

**return** **true**;

} System.***out***.println("\n=== TIME COMPLEXITY ANALYSIS ===");

System.***out***.println("Operation | Time Complexity | Explanation");

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

System.***out***.println("Add | O(1) | Insert at end of array");

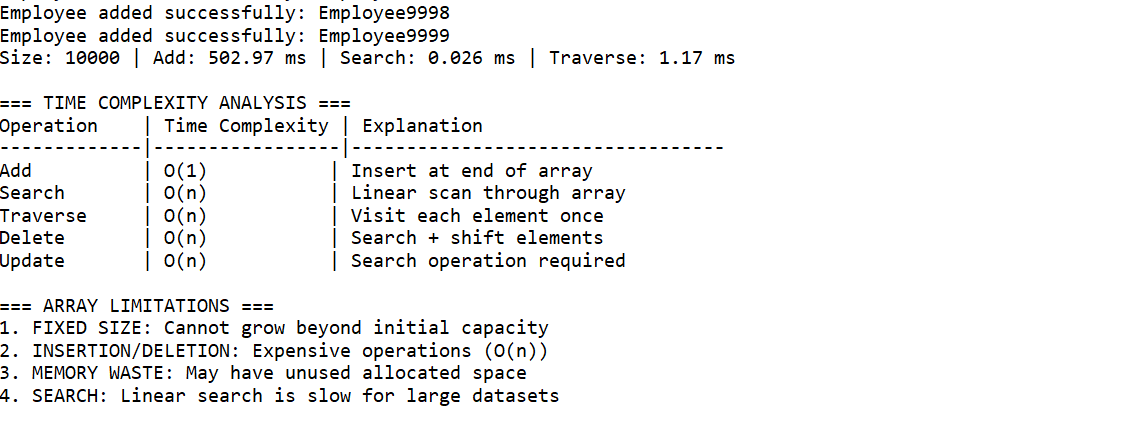
System.***out***.println("Search | O(n) | Linear scan through array");

System.***out***.println("Traverse | O(n) | Visit each element once");

System.***out***.println("Delete | O(n) | Search + shift elements");

System.***out***.println("Update | O(n) | Search operation required");

3.Output :



4.Time complexity

AddO(1)Insert at array end

SearchO(n)Must check each element

TraverseO(n)Visit every elementDeleteO(n)Search + shift remaining

UpdateO(n)Requires search first

**Exercise 5: Task Management System**

## **1.Linked List Types Explained:**

### **1. Singly Linked List (Implemented)**

[Task1] -> [Task2] -> [Task3] -> null

* Each node points to next node only
* Forward traversal only
* Memory efficient

### **2. Doubly Linked List**

null <- [Task1] <-> [Task2] <-> [Task3] -> null

* Each node has prev and next pointers
* Bidirectional traversal
* More memory overhead but faster deletions

### **3. Circular Linked List**

[Task1] -> [Task2] -> [Task3] -> [Task1]

* Last node points back to first
* Useful for round-robin scheduling
* No null termination

2.Code:

Task.java

**package** TaskManage;

**class** Task {

**private** **int** taskId;

**private** String taskName;

**private** String status; // "Pending", "In Progress", "Completed"

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

**this**.taskId = taskId;

**this**.taskName = taskName;

**this**.status = status;

}

// Getters

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

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

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

// Setters

**public** **void** setTaskName(String taskName) { **this**.taskName = taskName; }

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

@Override

**public** String toString() {

**return** String.*format*("Task{ID=%d, Name='%s', Status='%s'}",

taskId, taskName, status);

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj) **return** **true**;

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

Task task = (Task) obj;

**return** taskId == task.taskId;

}

}

/\*\*

\* Node class for the linked list

\* Each node contains a Task and reference to next node

\*/

**class** TaskNode {

Task task;

TaskNode next;

**public** TaskNode(Task task) {

**this**.task = task;

**this**.next = **null**;

}

}

TaskManagementSystem.java

**package** TaskManage;

**public** **class** TaskManagementSystem {

**private** TaskNode head;

**private** **int** size;

**public** TaskManagementSystem() {

**this**.head = **null**;

**this**.size = 0;

}

/\*\*

\* ADD TASK AT BEGINNING

\* Time Complexity: O(1) - Constant time insertion

\* Space Complexity: O(1) - Only one new node created

\*

\* Process: Create new node and make it point to current head

\*/

**public** **void** addTask(Task task) {

// Check for duplicate task ID

**if** (searchTask(task.getTaskId()) != **null**) {

System.***out***.println("Error: Task ID " + task.getTaskId() + " already exists!");

**return**;

}

TaskNode newNode = **new** TaskNode(task);

newNode.next = head;

head = newNode;

size++;

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

}

/\*\*

\* ADD TASK AT END

\* Time Complexity: O(n) - Must traverse to end

\* Space Complexity: O(1) - Only one new node created

\*

\* Process: Traverse to last node and link new node

\*/

**public** **void** addTaskAtEnd(Task task) {

**if** (searchTask(task.getTaskId()) != **null**) {

System.***out***.println("Error: Task ID " + task.getTaskId() + " already exists!");

**return**;

}

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

\* Time Complexity: O(n) - Linear search through list

\* Space Complexity: O(1) - No additional space needed

\*

\* Process: Traverse list until task found or end reached

\*/

**public** Task searchTask(**int** taskId) {

TaskNode current = head;

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

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

**return** current.task;

}

current = current.next;

}

**return** **null**; // Task not found

}

/\*\*

\* SEARCH TASK BY NAME

\* Time Complexity: O(n) - Linear search through list

\* Space Complexity: O(1) - No additional space needed

\*/

**public** Task searchTaskByName(String taskName) {

TaskNode current = head;

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

**if** (current.task.getTaskName().equalsIgnoreCase(taskName)) {

**return** current.task;

}

current = current.next;

}

**return** **null**;

}

/\*\*

\* TRAVERSE ALL TASKS

\* Time Complexity: O(n) - Must visit each node once

\* Space Complexity: O(1) - No additional space needed

\*

\* Process: Follow next pointers from head to end

\*/

**public** **void** traverseTasks() {

**if** (head == **null**) {

System.***out***.println("No tasks in the system.");

**return**;

}

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

TaskNode current = head;

**int** position = 1;

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

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

current = current.next;

position++;

}

System.***out***.println("Total tasks: " + size);

}

/\*\*

\* DELETE TASK BY ID

\* Time Complexity: O(n) - Linear search to find task

\* Space Complexity: O(1) - No additional space needed

\*

\* Process: Find task and adjust pointers to skip deleted node

\*/

**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) {

String deletedName = head.task.getTaskName();

head = head.next;

size--;

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

**return** **true**;

}

// Search for task in the rest of the list

TaskNode current = head;

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

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

String deletedName = current.next.task.getTaskName();

current.next = current.next.next; // Skip the node to delete

size--;

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

**return** **true**;

}

current = current.next;

}

System.***out***.println("Task with ID " + taskId + " not found!");

**return** **false**;

}

// Utility methods

**public** **int** getSize() { **return** size; }

**public** **boolean** isEmpty() { **return** head == **null**; }

/\*\*

\* DEMONSTRATION AND TESTING

\*/

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

System.***out***.println("=== TASK MANAGEMENT SYSTEM DEMONSTRATION ===\n");

TaskManagementSystem tms = **new** TaskManagementSystem();

// Add sample tasks

System.***out***.println("--- ADDING TASKS ---");

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

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

tms.addTask(**new** Task(103, "Create API Endpoints", "Pending"));

tms.addTaskAtEnd(**new** Task(104, "Write Unit Tests", "Pending"));

tms.addTaskAtEnd(**new** Task(105, "Deploy to Production", "Completed"));

tms.addTask(**new** Task(106, "Code Review", "In Progress"));

// Traverse all tasks

tms.traverseTasks();

// Search operations

System.***out***.println("\n--- SEARCH OPERATIONS ---");

Task found = tms.searchTask(103);

**if** (found != **null**) {

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

}

Task foundByName = tms.searchTaskByName("Write Unit Tests");

**if** (foundByName != **null**) {

System.***out***.println("Found by name: " + foundByName);

}

// Delete task

System.***out***.println("\n--- DELETE OPERATIONS ---");

tms.deleteTask(102);

tms.traverseTasks();

// Performance analysis

*performanceAnalysis*();

}

/\*\*

\* PERFORMANCE ANALYSIS COMPARING OPERATIONS

\*/

**public** **static** **void** performanceAnalysis() {

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

**int**[] sizes = {100, 1000, 5000};

**for** (**int** size : sizes) {

TaskManagementSystem tms = **new** TaskManagementSystem();

// Add operations

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

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

tms.addTask(**new** Task(i, "Task" + i, "Pending"));

}

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

// Search operations

startTime = System.*nanoTime*();

tms.searchTask(size / 2); // Search middle element

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

// Traverse operations

startTime = System.*nanoTime*();

TaskNode current = tms.head;

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

current = current.next;

}

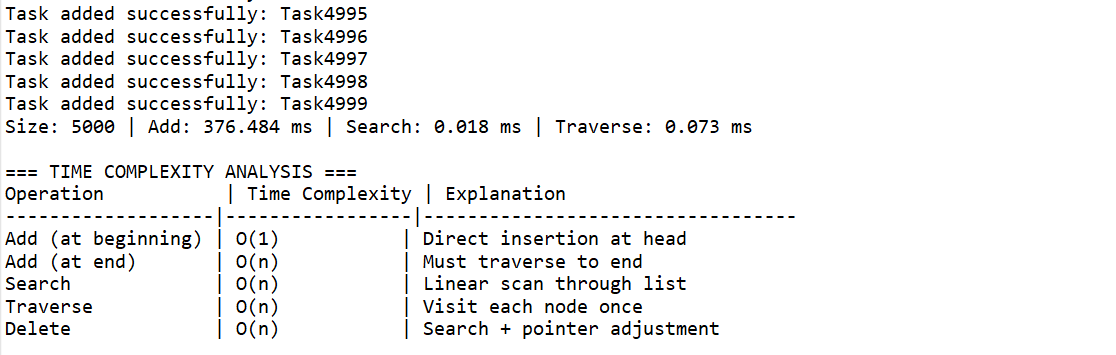
**long** traverseTime = System.*nanoTime*() - startTime;

System.***out***.printf("Size: %d | Add: %.3f ms | Search: %.3f ms | Traverse: %.3f ms\n",

size, addTime/1\_000\_000.0, searchTime/1\_000\_000.0, traverseTime/1\_000\_000.0);

}

3.Output:



4.Time complexity Analysis :

Add (beginning)O(1)Direct head insertion

Add (end)O(n)Must traverse to end

SearchO(n)Linear scan required

DeleteO(n)Search + pointer update

TraverseO(n)Visit each node once

**Exercise 6: Library Management System**

1.Explanation of linear and binary search

**Linear Search:**

* Examines each element in the collection sequentially from start to finish
* Works on both sorted and unsorted data
* Time complexity: O(n) - worst case checks every element
* Space complexity: O(1) - uses constant extra space

**Binary Search:**

* Divides the search space in half with each comparison
* Requires data to be sorted beforehand
* Time complexity: O(log n) - much faster for large datasets
* Space complexity: O(1) for iterative, O(log n) for recursive implementation

2.Code:

Book.java

**package** LibraryManagementSys;

**class** Book **implements** Comparable<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; }

// Setters

**public** **void** setBookId(**int** bookId) { **this**.bookId = bookId; }

**public** **void** setTitle(String title) { **this**.title = title; }

**public** **void** setAuthor(String author) { **this**.author = author; }

@Override

**public** **int** compareTo(Book other) {

**return** **this**.title.compareToIgnoreCase(other.title);

}

@Override

**public** String toString() {

**return** String.*format*("Book{ID: %d, Title: '%s', Author: '%s'}",

bookId, title, author);

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj) **return** **true**;

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

Book book = (Book) obj;

**return** bookId == book.bookId;

}

@Override

**public** **int** hashCode() {

**return** Objects.hash(bookId);

}

}

LibraryManager.java

**package** LibraryManagementSys;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.HashSet;

**import** java.util.List;

**import** java.util.Set;

**class** LibraryManager {

**private** List<Book> books;

**private** List<Book> sortedBooksByTitle;

**private** **boolean** isSorted;

**public** LibraryManager() {

**this**.books = **new** ArrayList<>();

**this**.sortedBooksByTitle = **new** ArrayList<>();

**this**.isSorted = **false**;

}

/\*\*

\* Add a book to the library

\*/

**public** **void** addBook(Book book) {

books.add(book);

isSorted = **false**; // Mark as unsorted when new book is added

}

/\*\*

\* Sort books by title for binary search

\*/

**public** **void** sortBooksByTitle() {

sortedBooksByTitle = **new** ArrayList<>(books);

Collections.*sort*(sortedBooksByTitle);

isSorted = **true**;

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

}

/\*\*

\* Linear Search Implementation

\* Time Complexity: O(n)

\* Space Complexity: O(k) where k is number of matches

\*/

**public** List<Book> linearSearchByTitle(String title) {

List<Book> results = **new** ArrayList<>();

**int** comparisons = 0;

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

**for** (Book book : books) {

comparisons++;

**if** (book.getTitle().equalsIgnoreCase(title)) {

results.add(book);

}

}

**long** endTime = System.*nanoTime*();

**long** duration = endTime - startTime;

System.***out***.printf("Linear Search - Comparisons: %d, Time: %d ns%n",

comparisons, duration);

**return** results;

}

/\*\*

\* Binary Search Implementation (Exact Match)

\* Time Complexity: O(log n)

\* Space Complexity: O(1)

\*/

**public** Book binarySearchByTitle(String title) {

**if** (!isSorted) {

sortBooksByTitle();

}

**int** comparisons = 0;

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

**int** left = 0;

**int** right = sortedBooksByTitle.size() - 1;

Book result = **null**;

**while** (left <= right) {

comparisons++;

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

Book midBook = sortedBooksByTitle.get(mid);

**int** comparison = title.compareToIgnoreCase(midBook.getTitle());

**if** (comparison == 0) {

result = midBook;

**break**;

} **else** **if** (comparison < 0) {

right = mid - 1;

} **else** {

left = mid + 1;

}

}

**long** endTime = System.*nanoTime*();

**long** duration = endTime - startTime;

System.***out***.printf("Binary Search - Comparisons: %d, Time: %d ns%n",

comparisons, duration);

**return** result;

}

/\*\*

\* Binary Search to find all books with matching title

\* Since there might be duplicates, we need to find the range

\*/

**public** List<Book> binarySearchAllByTitle(String title) {

**if** (!isSorted) {

sortBooksByTitle();

}

List<Book> results = **new** ArrayList<>();

Book found = binarySearchByTitle(title);

**if** (found != **null**) {

// Find all books with the same title

**for** (Book book : sortedBooksByTitle) {

**if** (book.getTitle().equalsIgnoreCase(title)) {

results.add(book);

}

}

}

**return** results;

}

/\*\*

\* Linear search by author

\*/

**public** List<Book> linearSearchByAuthor(String author) {

List<Book> results = **new** ArrayList<>();

**for** (Book book : books) {

**if** (book.getAuthor().equalsIgnoreCase(author)) {

results.add(book);

}

}

**return** results;

}

/\*\*

\* Display all books in the library

\*/

**public** **void** displayAllBooks() {

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

**if** (books.isEmpty()) {

System.***out***.println("No books in the library.");

} **else** {

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

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

}

}

System.***out***.println();

}

/\*\*

\* Display sorted books

\*/

**public** **void** displaySortedBooks() {

**if** (!isSorted) {

sortBooksByTitle();

}

System.***out***.println("\n=== Books Sorted by Title ===");

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

System.***out***.println((i + 1) + ". " + sortedBooksByTitle.get(i));

}

System.***out***.println();

}

/\*\*

\* Get library statistics

\*/

**public** **void** displayStatistics() {

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

System.***out***.println("Total books: " + books.size());

System.***out***.println("Is sorted: " + (isSorted ? "Yes" : "No"));

// Count unique authors

Set<String> uniqueAuthors = **new** HashSet<>();

**for** (Book book : books) {

uniqueAuthors.add(book.getAuthor().toLowerCase());

}

System.***out***.println("Unique authors: " + uniqueAuthors.size());

System.***out***.println();

}

**public** **int** getBookCount() {

**return** books.size();

}

}

SearchPerformanceAnalyser.java

**class** SearchPerformanceAnalyzer {

**public** **static** **void** compareSearchPerformance(LibraryManager library, String searchTitle) {

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

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

System.***out***.println();

// Linear Search

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

List<Book> linearResults = library.linearSearchByTitle(searchTitle);

System.***out***.println("Found " + linearResults.size() + " book(s)");

System.***out***.println();

// Binary Search

System.***out***.println("Binary Search Results:");

List<Book> binaryResults = library.binarySearchAllByTitle(searchTitle);

System.***out***.println("Found " + binaryResults.size() + " book(s)");

System.***out***.println();

}

**public** **static** **void** analyzeComplexity(**int** dataSize) {

System.***out***.println("\n=== Time Complexity Analysis ===");

System.***out***.println("Dataset size: " + dataSize + " books");

System.***out***.println();

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

System.***out***.println("- Best case: O(1) - item found at first position");

System.***out***.println("- Average case: O(n/2) - item found in middle");

System.***out***.println("- Worst case: O(n) - item at end or not found");

System.***out***.printf("- For %d books, worst case = %d comparisons%n", dataSize, dataSize);

System.***out***.println();

System.***out***.println("Binary Search:");

System.***out***.println("- Best case: O(1) - item found at middle");

System.***out***.println("- Average case: O(log n)");

System.***out***.println("- Worst case: O(log n) - item not found");

**int** binaryWorstCase = (**int**) Math.*ceil*(Math.*log*(dataSize) / Math.*log*(2));

System.***out***.printf("- For %d books, worst case = %d comparisons%n", dataSize, binaryWorstCase);

System.***out***.println();

System.***out***.println("Performance Advantage:");

**if** (dataSize > 1) {

**double** advantage = (**double**) dataSize / binaryWorstCase;

System.***out***.printf("Binary search is approximately %.1fx faster than linear search%n", advantage);

}

System.***out***.println();

}

}

LibraryManagementSystem.java

**package** LibraryManagementSys;

**import** java.util.List;

**import** java.util.Scanner;

**public** **class** LibraryManagementSystem {

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

LibraryManager library = **new** LibraryManager();

// Sample data

*populateLibrary*(library);

// Display all books

library.displayAllBooks();

// Display statistics

library.displayStatistics();

// Sort books for binary search

library.sortBooksByTitle();

library.displaySortedBooks();

// Demonstrate search algorithms

*demonstrateSearches*(library);

// Performance analysis

SearchPerformanceAnalyzer.*analyzeComplexity*(library.getBookCount());

// Interactive demo

*runInteractiveDemo*(library);

}

**private** **static** **void** populateLibrary(LibraryManager library) {

// Adding sample books

library.addBook(**new** Book(1, "The Great Gatsby", "F. Scott Fitzgerald"));

library.addBook(**new** Book(2, "To Kill a Mockingbird", "Harper Lee"));

library.addBook(**new** Book(3, "1984", "George Orwell"));

library.addBook(**new** Book(4, "Pride and Prejudice", "Jane Austen"));

library.addBook(**new** Book(5, "The Catcher in the Rye", "J.D. Salinger"));

library.addBook(**new** Book(6, "Animal Farm", "George Orwell"));

library.addBook(**new** Book(7, "Lord of the Flies", "William Golding"));

library.addBook(**new** Book(8, "Brave New World", "Aldous Huxley"));

library.addBook(**new** Book(9, "The Hobbit", "J.R.R. Tolkien"));

library.addBook(**new** Book(10, "Harry Potter and the Sorcerer's Stone", "J.K. Rowling"));

library.addBook(**new** Book(11, "The Great Gatsby", "F. Scott Fitzgerald")); // Duplicate for testing

library.addBook(**new** Book(12, "Fahrenheit 451", "Ray Bradbury"));

}

**private** **static** **void** demonstrateSearches(LibraryManager library) {

System.***out***.println("=== Search Demonstrations ===");

// Test 1: Search for existing book

SearchPerformanceAnalyzer.*compareSearchPerformance*(library, "1984");

// Test 2: Search for book with duplicates

SearchPerformanceAnalyzer.*compareSearchPerformance*(library, "The Great Gatsby");

// Test 3: Search for non-existing book

SearchPerformanceAnalyzer.*compareSearchPerformance*(library, "Non-existent Book");

// Test 4: Search by author

System.***out***.println("=== Search by Author ===");

List<Book> authorResults = library.linearSearchByAuthor("George Orwell");

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

**for** (Book book : authorResults) {

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

}

System.***out***.println();

}

**private** **static** **void** runInteractiveDemo(LibraryManager library) {

Scanner scanner = **new** Scanner(System.***in***);

System.***out***.println("=== Interactive Demo ===");

System.***out***.println("Choose an option:");

System.***out***.println("1. Search by title (Linear Search)");

System.***out***.println("2. Search by title (Binary Search)");

System.***out***.println("3. Search by author");

System.***out***.println("4. Add new book");

System.***out***.println("5. Display all books");

System.***out***.println("6. Exit");

**while** (**true**) {

System.***out***.print("\nEnter your choice (1-6): ");

**if** (!scanner.hasNextInt()) {

System.***out***.println("Please enter a valid number.");

scanner.next(); // consume invalid input

**continue**;

}

**int** choice = scanner.nextInt();

scanner.nextLine(); // consume newline

**switch** (choice) {

**case** 1:

System.***out***.print("Enter book title to search: ");

String titleLinear = scanner.nextLine();

List<Book> linearResults = library.linearSearchByTitle(titleLinear);

*displaySearchResults*(linearResults, "Linear Search");

**break**;

**case** 2:

System.***out***.print("Enter book title to search: ");

String titleBinary = scanner.nextLine();

List<Book> binaryResults = library.binarySearchAllByTitle(titleBinary);

*displaySearchResults*(binaryResults, "Binary Search");

**break**;

**case** 3:

System.***out***.print("Enter author name to search: ");

String author = scanner.nextLine();

List<Book> authorResults = library.linearSearchByAuthor(author);

*displaySearchResults*(authorResults, "Author Search");

**break**;

**case** 4:

System.***out***.print("Enter book ID: ");

**int** id = scanner.nextInt();

scanner.nextLine();

System.***out***.print("Enter book title: ");

String title = scanner.nextLine();

System.***out***.print("Enter author name: ");

String authorName = scanner.nextLine();

library.addBook(**new** Book(id, title, authorName));

System.***out***.println("Book added successfully!");

**break**;

**case** 5:

library.displayAllBooks();

**break**;

**case** 6:

System.***out***.println("Thank you for using the Library Management System!");

scanner.close();

**return**;

**default**:

System.***out***.println("Invalid choice. Please enter 1-6.");

}

}

}

**private** **static** **void** displaySearchResults(List<Book> results, String searchType) {

System.***out***.println("\n" + searchType + " Results:");

**if** (results.isEmpty()) {

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

} **else** {

System.***out***.println("Found " + results.size() + " book(s):");

**for** (Book book : results) {

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

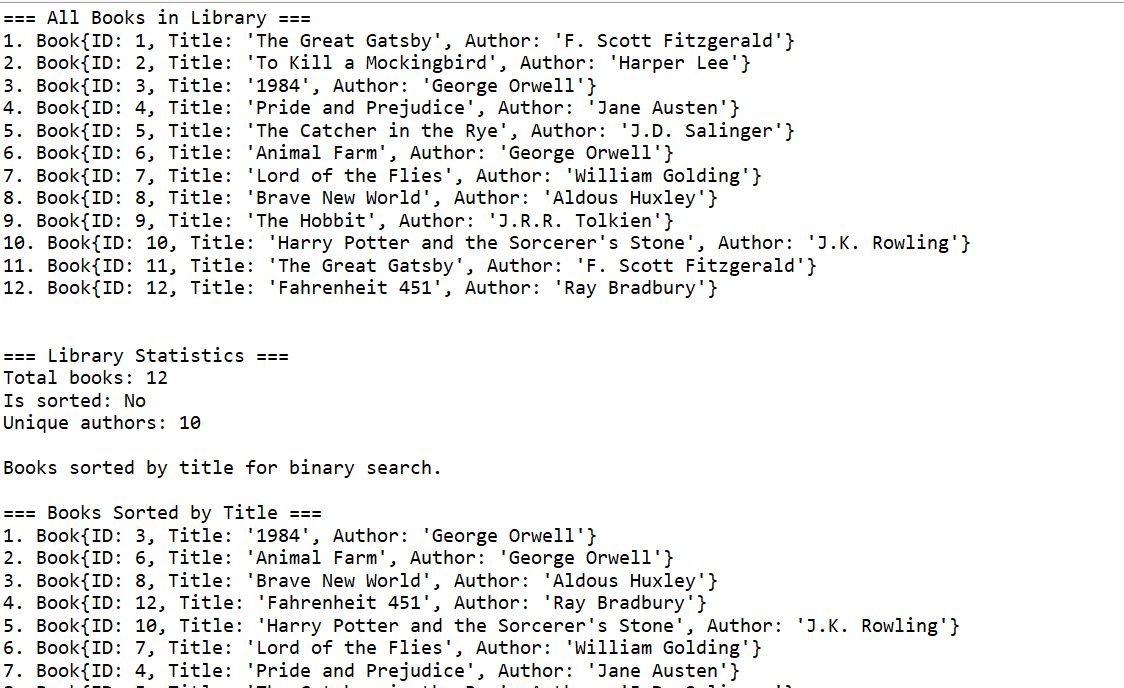
}

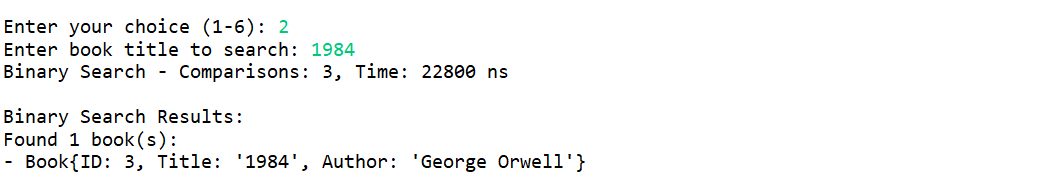
}

}

}

3.Output:





### **4.Time Complexity Comparison**

**Linear Search:**

* **Best Case:** O(1) - element found at first position
* **Average Case:** O(n/2) - element found in the middle
* **Worst Case:** O(n) - element at end or not found

**Binary Search:**

* **Best Case:** O(1) - element found at middle position
* **Average Case:** O(log n)
* **Worst Case:** O(log n) - element not found

**Exercise 7: Financial Forecasting**

**1.Recursion** is a programming technique where a function calls itself to solve smaller instances of the same problem. It consists of:

* **Base Case:** The stopping condition that prevents infinite recursion
* **Recursive Case:** The function calling itself with modified parameters

2.Code :

FinancialDataPoint.java

**package** FinanceForecast;

**import** java.text.DecimalFormat;

/\*\*

\* Represents historical financial data point

\*/

**class** FinancialDataPoint {

**private** **int** period;

**private** **double** value;

**private** **double** growthRate;

**public** FinancialDataPoint(**int** period, **double** value) {

**this**.period = period;

**this**.value = value;

**this**.growthRate = 0.0;

}

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

**this**.period = period;

**this**.value = value;

**this**.growthRate = growthRate;

}

// Getters and Setters

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

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

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

**public** **void** setPeriod(**int** period) { **this**.period = period; }

**public** **void** setValue(**double** value) { **this**.value = value; }

**public** **void** setGrowthRate(**double** growthRate) { **this**.growthRate = growthRate; }

@Override

**public** String toString() {

DecimalFormat df = **new** DecimalFormat("#,##0.00");

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

period, df.format(value), growthRate \* 100);

}

}

Financialforecast.java

**package** FinanceForecast;

**import** java.text.DecimalFormat;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.HashMap;

**import** java.util.List;

**import** java.util.Map;

**import** java.util.Random;

**class** FinancialForecaster {

**private** List<FinancialDataPoint> historicalData;

**private** Map<String, Double> memoizationCache;

**private** **int** recursionCallCount;

**private** **int** memoizedCallCount;

**public** FinancialForecaster() {

**this**.historicalData = **new** ArrayList<>();

**this**.memoizationCache = **new** HashMap<>();

**this**.recursionCallCount = 0;

**this**.memoizedCallCount = 0;

}

/\*\*

\* Add historical data point

\*/

**public** **void** addHistoricalData(**int** period, **double** value) {

historicalData.add(**new** FinancialDataPoint(period, value));

calculateGrowthRates();

}

/\*\*

\* Calculate growth rates for historical data

\*/

**private** **void** calculateGrowthRates() {

**if** (historicalData.size() < 2) **return**;

**for** (**int** i = 1; i < historicalData.size(); i++) {

FinancialDataPoint current = historicalData.get(i);

FinancialDataPoint previous = historicalData.get(i - 1);

**double** growthRate = (current.getValue() - previous.getValue()) / previous.getValue();

current.setGrowthRate(growthRate);

}

}

/\*\*

\* Basic Recursive Future Value Calculation

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

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

\*/

**public** **double** calculateFutureValueRecursive(**double** presentValue, **double** growthRate, **int** periods) {

recursionCallCount++;

// Base case

**if** (periods == 0) {

**return** presentValue;

}

// Recursive case

**return** calculateFutureValueRecursive(presentValue \* (1 + growthRate), growthRate, periods - 1);

}

/\*\*

\* Optimized Recursive Future Value with Memoization

\* Time Complexity: O(n) with memoization, but avoids redundant calculations

\*/

**public** **double** calculateFutureValueMemoized(**double** presentValue, **double** growthRate, **int** periods) {

String key = presentValue + ":" + growthRate + ":" + periods;

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

memoizedCallCount++;

**return** memoizationCache.get(key);

}

recursionCallCount++;

**double** result;

**if** (periods == 0) {

result = presentValue;

} **else** {

result = calculateFutureValueMemoized(presentValue \* (1 + growthRate), growthRate, periods - 1);

}

memoizationCache.put(key, result);

**return** result;

}

/\*\*

\* Recursive Compound Interest Calculator

\* Accounts for different compounding frequencies

\*/

**public** **double** calculateCompoundInterestRecursive(**double** principal, **double** annualRate,

**int** compoundingPerYear, **int** years) {

**return** calculateCompoundInterestHelper(principal, annualRate / compoundingPerYear,

compoundingPerYear \* years);

}

**private** **double** calculateCompoundInterestHelper(**double** amount, **double** ratePerPeriod, **int** periods) {

recursionCallCount++;

**if** (periods == 0) {

**return** amount;

}

**return** calculateCompoundInterestHelper(amount \* (1 + ratePerPeriod), ratePerPeriod, periods - 1);

}

/\*\*

\* Recursive Prediction Based on Historical Growth Patterns

\* Uses weighted average of historical growth rates

\*/

**public** List<FinancialDataPoint> predictFutureValuesRecursive(**int** futurePeriods) {

**if** (historicalData.isEmpty()) {

**throw** **new** IllegalStateException("No historical data available for prediction");

}

**double** averageGrowthRate = calculateWeightedAverageGrowthRate();

**double** lastValue = historicalData.get(historicalData.size() - 1).getValue();

**return** predictFutureValuesHelper(lastValue, averageGrowthRate, futurePeriods,

historicalData.size(), **new** ArrayList<>());

}

**private** List<FinancialDataPoint> predictFutureValuesHelper(**double** currentValue, **double** growthRate,

**int** remainingPeriods, **int** currentPeriod,

List<FinancialDataPoint> predictions) {

recursionCallCount++;

// Base case

**if** (remainingPeriods == 0) {

**return** predictions;

}

// Calculate next value

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

predictions.add(**new** FinancialDataPoint(currentPeriod + 1, nextValue, growthRate));

// Recursive case

**return** predictFutureValuesHelper(nextValue, growthRate, remainingPeriods - 1,

currentPeriod + 1, predictions);

}

/\*\*

\* Calculate weighted average growth rate (recent data has more weight)

\*/

**private** **double** calculateWeightedAverageGrowthRate() {

**if** (historicalData.size() < 2) **return** 0.0;

**double** weightedSum = 0.0;

**double** totalWeight = 0.0;

**for** (**int** i = 1; i < historicalData.size(); i++) {

**double** weight = i; // More recent data gets higher weight

**double** growthRate = historicalData.get(i).getGrowthRate();

weightedSum += growthRate \* weight;

totalWeight += weight;

}

**return** weightedSum / totalWeight;

}

/\*\*

\* Recursive Monte Carlo Simulation for Risk Analysis

\* Simulates multiple future scenarios with varying growth rates

\*/

**public** Map<String, Double> performMonteCarloSimulation(**int** periods, **int** simulations,

**double** volatility) {

**if** (historicalData.isEmpty()) {

**throw** **new** IllegalStateException("No historical data available for simulation");

}

**double** baseGrowthRate = calculateWeightedAverageGrowthRate();

**double** startValue = historicalData.get(historicalData.size() - 1).getValue();

List<Double> finalValues = **new** ArrayList<>();

Random random = **new** Random();

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

**double** simulatedValue = monteCarloRecursive(startValue, baseGrowthRate,

volatility, periods, random);

finalValues.add(simulatedValue);

}

**return** calculateStatistics(finalValues);

}

**private** **double** monteCarloRecursive(**double** currentValue, **double** baseGrowthRate,

**double** volatility, **int** remainingPeriods, Random random) {

recursionCallCount++;

**if** (remainingPeriods == 0) {

**return** currentValue;

}

// Add random variation to growth rate

**double** randomFactor = (random.nextGaussian() \* volatility);

**double** adjustedGrowthRate = baseGrowthRate + randomFactor;

**double** nextValue = currentValue \* (1 + adjustedGrowthRate);

**return** monteCarloRecursive(nextValue, baseGrowthRate, volatility,

remainingPeriods - 1, random);

}

**private** Map<String, Double> calculateStatistics(List<Double> values) {

Collections.*sort*(values);

Map<String, Double> stats = **new** HashMap<>();

**double** sum = values.stream().mapToDouble(Double::doubleValue).sum();

**double** mean = sum / values.size();

**double** variance = values.stream()

.mapToDouble(val -> Math.*pow*(val - mean, 2))

.sum() / values.size();

stats.put("mean", mean);

stats.put("median", values.get(values.size() / 2));

stats.put("min", values.get(0));

stats.put("max", values.get(values.size() - 1));

stats.put("standardDeviation", Math.*sqrt*(variance));

stats.put("percentile\_5", values.get((**int**)(values.size() \* 0.05)));

stats.put("percentile\_95", values.get((**int**)(values.size() \* 0.95)));

**return** stats;

}

/\*\*

\* Non-recursive iterative solution for comparison

\*/

**public** **double** calculateFutureValueIterative(**double** presentValue, **double** growthRate, **int** periods) {

**double** result = presentValue;

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

result \*= (1 + growthRate);

}

**return** result;

}

/\*\*

\* Mathematical formula solution (most efficient)

\*/

**public** **double** calculateFutureValueFormula(**double** presentValue, **double** growthRate, **int** periods) {

**return** presentValue \* Math.*pow*(1 + growthRate, periods);

}

// Utility methods for performance tracking and display

**public** **void** resetCounters() {

recursionCallCount = 0;

memoizedCallCount = 0;

memoizationCache.clear();

}

**public** **int** getRecursionCallCount() { **return** recursionCallCount; }

**public** **int** getMemoizedCallCount() { **return** memoizedCallCount; }

**public** **void** displayHistoricalData() {

System.***out***.println("\n=== Historical Data ===");

**if** (historicalData.isEmpty()) {

System.***out***.println("No historical data available.");

} **else** {

DecimalFormat df = **new** DecimalFormat("#,##0.00");

**for** (FinancialDataPoint point : historicalData) {

System.***out***.println(point);

}

}

System.***out***.println();

}

}

PerformanceAnalyser.java

**package** FinanceForecast;

**import** java.text.DecimalFormat;

**class** PerformanceAnalyzer {

**public** **static** **void** compareCalculationMethods(FinancialForecaster forecaster,

**double** presentValue, **double** growthRate, **int** periods) {

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

System.***out***.printf("Calculating future value: PV=$%.2f, Rate=%.2f%%, Periods=%d%n",

presentValue, growthRate \* 100, periods);

System.***out***.println();

// Reset counters

forecaster.resetCounters();

// Recursive method

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

**double** recursiveResult = forecaster.calculateFutureValueRecursive(presentValue, growthRate, periods);

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

**int** recursiveCalls = forecaster.getRecursionCallCount();

// Reset for memoized version

forecaster.resetCounters();

// Memoized recursive method

startTime = System.*nanoTime*();

**double** memoizedResult = forecaster.calculateFutureValueMemoized(presentValue, growthRate, periods);

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

**int** memoizedCalls = forecaster.getRecursionCallCount();

**int** cacheHits = forecaster.getMemoizedCallCount();

// Iterative method

startTime = System.*nanoTime*();

**double** iterativeResult = forecaster.calculateFutureValueIterative(presentValue, growthRate, periods);

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

// Formula method

startTime = System.*nanoTime*();

**double** formulaResult = forecaster.calculateFutureValueFormula(presentValue, growthRate, periods);

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

// Display results

DecimalFormat df = **new** DecimalFormat("#,##0.00");

System.***out***.println("Results:");

System.***out***.printf("Recursive: $%s (Time: %,d ns, Calls: %d)%n",

df.format(recursiveResult), recursiveTime, recursiveCalls);

System.***out***.printf("Memoized: $%s (Time: %,d ns, Calls: %d, Cache hits: %d)%n",

df.format(memoizedResult), memoizedTime, memoizedCalls, cacheHits);

System.***out***.printf("Iterative: $%s (Time: %,d ns)%n",

df.format(iterativeResult), iterativeTime);

System.***out***.printf("Formula: $%s (Time: %,d ns)%n",

df.format(formulaResult), formulaTime);

System.***out***.println("\nTime Complexity Analysis:");

System.***out***.println("- Recursive: O(n) time, O(n) space (call stack)");

System.***out***.println("- Memoized: O(n) time, O(n) space (cache + call stack)");

System.***out***.println("- Iterative: O(n) time, O(1) space");

System.***out***.println("- Formula: O(1) time, O(1) space");

System.***out***.println();

}

}

Financialforecastingtool.java

**package** FinanceForecast;

**import** java.text.DecimalFormat;

**import** java.util.List;

**import** java.util.Map;

**import** java.util.Scanner;

**public** **class** FinancialForecastingTool {

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

FinancialForecaster forecaster = **new** FinancialForecaster();

// Setup sample data

*setupSampleData*(forecaster);

// Display historical data

forecaster.displayHistoricalData();

// Demonstrate different calculation methods

*demonstrateCalculationMethods*(forecaster);

// Perform predictions

*demonstratePredictions*(forecaster);

// Monte Carlo simulation

*demonstrateMonteCarloSimulation*(forecaster);

// Interactive demo

*runInteractiveDemo*(forecaster);

}

**private** **static** **void** setupSampleData(FinancialForecaster forecaster) {

System.***out***.println("Setting up sample investment data...");

// Sample investment data over 10 years

forecaster.addHistoricalData(1, 10000.00); // Initial investment

forecaster.addHistoricalData(2, 10800.00); // 8% growth

forecaster.addHistoricalData(3, 11664.00); // 8% growth

forecaster.addHistoricalData(4, 12597.12); // 8% growth

forecaster.addHistoricalData(5, 13004.75); // 3.2% growth (market downturn)

forecaster.addHistoricalData(6, 14305.23); // 10% recovery

forecaster.addHistoricalData(7, 15725.75); // 9.9% growth

forecaster.addHistoricalData(8, 17013.42); // 8.2% growth

forecaster.addHistoricalData(9, 18374.70); // 8% growth

forecaster.addHistoricalData(10, 19844.68); // 8% growth

}

**private** **static** **void** demonstrateCalculationMethods(FinancialForecaster forecaster) {

System.***out***.println("=== Calculation Method Demonstrations ===");

// Simple future value calculation

PerformanceAnalyzer.*compareCalculationMethods*(forecaster, 10000, 0.08, 10);

// Compound interest calculation

System.***out***.println("=== Compound Interest Calculation ===");

**double** principal = 10000;

**double** annualRate = 0.08;

**int** compoundingFreq = 12; // Monthly

**int** years = 5;

forecaster.resetCounters();

**double** compoundResult = forecaster.calculateCompoundInterestRecursive(

principal, annualRate, compoundingFreq, years);

DecimalFormat df = **new** DecimalFormat("#,##0.00");

System.***out***.printf("Compound Interest Result: $%s%n", df.format(compoundResult));

System.***out***.printf("Recursive calls made: %d%n", forecaster.getRecursionCallCount());

System.***out***.println();

}

**private** **static** **void** demonstratePredictions(FinancialForecaster forecaster) {

System.***out***.println("=== Future Value Predictions ===");

forecaster.resetCounters();

List<FinancialDataPoint> predictions = forecaster.predictFutureValuesRecursive(5);

System.***out***.println("Predicted future values based on historical data:");

**for** (FinancialDataPoint prediction : predictions) {

System.***out***.println(prediction);

}

System.***out***.printf("Total recursive calls for prediction: %d%n",

forecaster.getRecursionCallCount());

System.***out***.println();

}

**private** **static** **void** demonstrateMonteCarloSimulation(FinancialForecaster forecaster) {

System.***out***.println("=== Monte Carlo Risk Analysis ===");

forecaster.resetCounters();

Map<String, Double> stats = forecaster.performMonteCarloSimulation(5, 1000, 0.15);

DecimalFormat df = **new** DecimalFormat("#,##0.00");

System.***out***.println("Monte Carlo Simulation Results (1000 simulations, 5 years):");

System.***out***.printf("Mean: $%s%n", df.format(stats.get("mean")));

System.***out***.printf("Median: $%s%n", df.format(stats.get("median")));

System.***out***.printf("5th Percentile: $%s%n", df.format(stats.get("percentile\_5")));

System.***out***.printf("95th Percentile: $%s%n", df.format(stats.get("percentile\_95")));

System.***out***.printf("Standard Deviation: $%s%n", df.format(stats.get("standardDeviation")));

System.***out***.printf("Min: $%s%n", df.format(stats.get("min")));

System.***out***.printf("Max: $%s%n", df.format(stats.get("max")));

System.***out***.printf("Total recursive calls for simulation: %d%n",

forecaster.getRecursionCallCount());

System.***out***.println();

}

**private** **static** **void** runInteractiveDemo(FinancialForecaster forecaster) {

Scanner scanner = **new** Scanner(System.***in***);

DecimalFormat df = **new** DecimalFormat("#,##0.00");

System.***out***.println("=== Interactive Financial Forecasting Demo ===");

System.***out***.println("Choose an option:");

System.***out***.println("1. Calculate future value");

System.***out***.println("2. Calculate compound interest");

System.***out***.println("3. Predict based on historical data");

System.***out***.println("4. Run Monte Carlo simulation");

System.***out***.println("5. Compare calculation methods");

System.***out***.println("6. Add historical data point");

System.***out***.println("7. Exit");

**while** (**true**) {

System.***out***.print("\nEnter your choice (1-7): ");

**if** (!scanner.hasNextInt()) {

System.***out***.println("Please enter a valid number.");

scanner.next();

**continue**;

}

**int** choice = scanner.nextInt();

**switch** (choice) {

**case** 1:

System.***out***.print("Enter present value: $");

**double** pv = scanner.nextDouble();

System.***out***.print("Enter growth rate (as decimal, e.g., 0.08 for 8%): ");

**double** rate = scanner.nextDouble();

System.***out***.print("Enter number of periods: ");

**int** periods = scanner.nextInt();

forecaster.resetCounters();

**double** fv = forecaster.calculateFutureValueRecursive(pv, rate, periods);

System.***out***.printf("Future Value: $%s%n", df.format(fv));

System.***out***.printf("Recursive calls: %d%n", forecaster.getRecursionCallCount());

**break**;

**case** 2:

System.***out***.print("Enter principal amount: $");

**double** principal = scanner.nextDouble();

System.***out***.print("Enter annual interest rate (as decimal): ");

**double** annualRate = scanner.nextDouble();

System.***out***.print("Enter compounding frequency per year: ");

**int** compFreq = scanner.nextInt();

System.***out***.print("Enter number of years: ");

**int** years = scanner.nextInt();

forecaster.resetCounters();

**double** compound = forecaster.calculateCompoundInterestRecursive(

principal, annualRate, compFreq, years);

System.***out***.printf("Compound Interest Result: $%s%n", df.format(compound));

System.***out***.printf("Recursive calls: %d%n", forecaster.getRecursionCallCount());

**break**;

**case** 3:

System.***out***.print("Enter number of future periods to predict: ");

**int** futurePeriods = scanner.nextInt();

**try** {

forecaster.resetCounters();

List<FinancialDataPoint> predictions =

forecaster.predictFutureValuesRecursive(futurePeriods);

System.***out***.println("Predictions:");

**for** (FinancialDataPoint prediction : predictions) {

System.***out***.println(prediction);

}

System.***out***.printf("Recursive calls: %d%n", forecaster.getRecursionCallCount());

} **catch** (IllegalStateException e) {

System.***out***.println("Error: " + e.getMessage());

}

**break**;

**case** 4:

System.***out***.print("Enter number of periods to simulate: ");

**int** simPeriods = scanner.nextInt();

System.***out***.print("Enter number of simulations: ");

**int** simCount = scanner.nextInt();

System.***out***.print("Enter volatility (e.g., 0.15 for 15%): ");

**double** volatility = scanner.nextDouble();

**try** {

forecaster.resetCounters();

Map<String, Double> stats = forecaster.performMonteCarloSimulation(

simPeriods, simCount, volatility);

System.***out***.println("Monte Carlo Results:");

System.***out***.printf("Mean: $%s%n", df.format(stats.get("mean")));

System.***out***.printf("Median: $%s%n", df.format(stats.get("median")));

System.***out***.printf("5th Percentile: $%s%n", df.format(stats.get("percentile\_5")));

System.***out***.printf("95th Percentile: $%s%n", df.format(stats.get("percentile\_95")));

System.***out***.printf("Recursive calls: %d%n", forecaster.getRecursionCallCount());

} **catch** (IllegalStateException e) {

System.***out***.println("Error: " + e.getMessage());

}

**break**;

**case** 5:

System.***out***.print("Enter present value: $");

**double** testPv = scanner.nextDouble();

System.***out***.print("Enter growth rate (as decimal): ");

**double** testRate = scanner.nextDouble();

System.***out***.print("Enter number of periods: ");

**int** testPeriods = scanner.nextInt();

PerformanceAnalyzer.*compareCalculationMethods*(

forecaster, testPv, testRate, testPeriods);

**break**;

**case** 6:

System.***out***.print("Enter period number: ");

**int** period = scanner.nextInt();

System.***out***.print("Enter value: $");

**double** value = scanner.nextDouble();

forecaster.addHistoricalData(period, value);

System.***out***.println("Historical data point added successfully!");

**break**;

**case** 7:

System.***out***.println("Thank you for using the Financial Forecasting Tool!");

scanner.close();

**return**;

**default**:

System.***out***.println("Invalid choice. Please enter 1-7.");

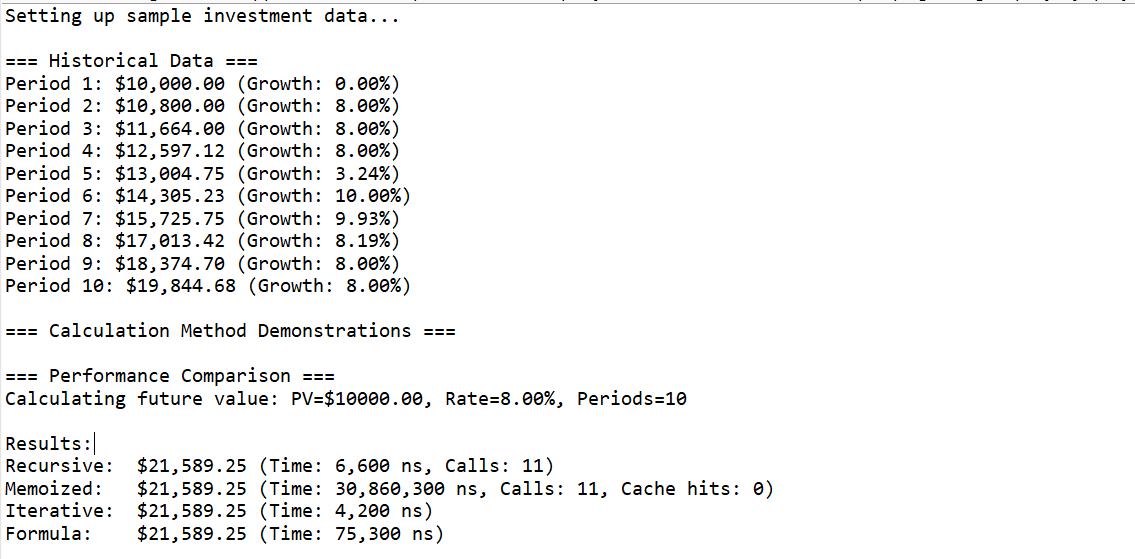
}

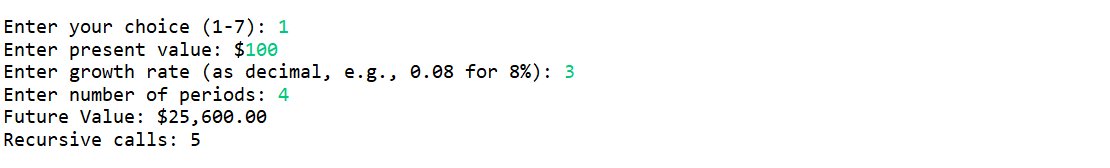
}

}

}

3.Output:





### **4.Time Complexity Analysis**

**Basic Recursive Future Value Calculation:**

* **Time Complexity:** O(n) where n is the number of periods
* **Space Complexity:** O(n) due to call stack depth
* **Problem:** Each recursive call creates a new stack frame