**Name - Moupiya Garai**

**Superset id - 6362737**

**Exercise 1: Inventory Management System**

### **Step 1:**

**Why Data Structures and Algorithms are Essential for Large Inventories:**

Data structures and algorithms are fundamental for inventory management because:

* **Efficiency**: Large warehouses may contain thousands or millions of products. Efficient data structures ensure quick access and modification of product information.
* **Scalability**: As inventory grows, the system must maintain performance without degradation.
* **Memory Management**: Proper data structures optimize memory usage, crucial for large datasets.
* **Real-time Operations**: Warehouse operations require instant updates for stock levels, pricing, and product information.

**Suitable Data Structures:**

* **HashMap**: Provides O(1) average time complexity for search, insert, and delete operations using product ID as key
* **ArrayList**: Good for sequential access but slower for searches (O(n))

### **Step 2:**

### InventoryManagementSystem :

* Product.java
* InventoryManager.java
* InventoryManagementSystem.java

**Step 3:**

**Implementation:**

**Product.java:**

### import java.util.HashMap;

### import java.util.Map;

### 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;

### }

### // Getter methods

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

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

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

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

### // Setter methods

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

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

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

### public String toString() {

### return "Product ID: " + productId + ", Name: " + productName +

### ", Quantity: " + quantity + ", Price: $" + String.format("%.2f", price);

### }

### }

### **InventoryManager.java:**

### import java.util.HashMap;

### import java.util.Map;

### class InventoryManager {

### private Map<String, Product> inventory;

### public InventoryManager() {

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

### }

### public void addProduct(Product product) {

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

### }

### public boolean updateProduct(String productId, String newProductName,

### int newQuantity, double newPrice) {

### if (inventory.containsKey(productId)) {

### Product product = inventory.get(productId);

### product.setProductName(newProductName);

### product.setQuantity(newQuantity);

### product.setPrice(newPrice);

### return true;

### }

### return false;

### }

### /\*

### Deletes a product from the inventory.

### return True if the product was deleted, false otherwise.

### \*/

### public boolean deleteProduct(String productId) {

### return inventory.remove(productId) != null;

### }

### public Product getProduct(String productId) {

### return inventory.get(productId);

### }

### public void displayAllProducts() {

### if (inventory.isEmpty()) {

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

### return;

### }

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

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

### }

### }

### }

**InventoryManagementSystem.java:**

class InventoryManagementSystem {

public static void main(String[] args) {

InventoryManager manager = new InventoryManager();

System.out.println("--- Adding Products ---");

manager.addProduct(new Product("P001", "Laptop", 10, 1200.00));

manager.addProduct(new Product("P002", "Mouse", 50, 25.50));

manager.addProduct(new Product("P003", "Keyboard", 30, 75.00));

manager.displayAllProducts();

System.out.println("\n--- Updating Product P002 ---");

manager.updateProduct("P002", "Gaming Mouse", 45, 35.00);

Product updatedMouse = manager.getProduct("P002");

if (updatedMouse != null) {

System.out.println("Updated: " + updatedMouse);

}

System.out.println("\n--- Getting Product P001 ---");

Product p001 = manager.getProduct("P001");

if (p001 != null) {

System.out.println("Found: " + p001);

}

System.out.println("\n--- Deleting Product P003 ---");

manager.deleteProduct("P003");

System.out.println("Inventory after deletion:");

manager.displayAllProducts();

System.out.println("--- Attempting to update a non-existent product ---\n");

boolean updatedNonExistent = manager.updateProduct("P004", "Monitor", 20, 300.00);

System.out.println("Updated non-existent P004: " + updatedNonExistent);

System.out.println("--- Final Inventory ---\n");

manager.displayAllProducts();

}

}

### Output:

### Step 4:

### Analysis:

**Time Complexity Analysis:**

* **Add Operation**: O(1) average case - HashMap provides constant time insertion
* **Update Operation**: O(1) average case - Direct access via key, then constant time property updates
* **Delete Operation**: O(1) average case - HashMap removal is constant time
* **Search Operation**: O(1) average case - HashMap lookup is constant time

**Optimization Strategies:**

1. **Load Factor Management**: Keep HashMap load factor optimal (default 0.75)
2. **Initial Capacity**: Set appropriate initial capacity to minimize rehashing
3. **Batch Operations**: Implement bulk operations for multiple products
4. **Caching**: Cache frequently accessed products in memory
5. **Indexing**: Create secondary indexes for searching by name or category

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

**Scenario:**

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

**Steps:**

### Step 1: Understand Asymptotic Notation

**Big O Notation Explained:** Big O notation describes the upper bound of algorithm performance, helping us understand how algorithms scale with input size.

**Common Time Complexities:**

* O(1) - Constant time
* O(log n) - Logarithmic time
* O(n) - Linear time
* O(n log n) - Linearithmic time
* O(n²) - Quadratic time

**Search Algorithm Scenarios:**

* **Best Case**: Element found immediately
* **Average Case**: Element found in middle of search space
* **Worst Case**: Element is last or not present

### Implementation

**SearchProduct.java:**

**class SearchProduct {**

**private String productId;**

**private String productName;**

**private String category;**

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

**this.productId = productId;**

**this.productName = productName;**

**this.category = category;**

**}**

**/\***

**Returns the product ID.**

**return The product ID.**

**\*/**

**public String getProductId() {**

**return productId;**

**}**

**/\***

**Returns the product name.**

**return The product name.**

**\*/**

**public String getProductName() {**

**return productName;**

**}**

**/\***

**Returns the product category.**

**return The product category.**

**\*/**

**public String getCategory() {**

**return category;**

**}**

**/\***

**Returns a string representation of the SearchProduct.**

**return A string containing product search details.**

**\*/**

**public String toString() {**

**return "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;**

**}**

**}**

**ProductSearcher.java:**

class ProductSearcher {

public SearchProduct linearSearch(SearchProduct[] products, String searchName) {

for (SearchProduct product : products) {

if (product.getProductName().equalsIgnoreCase(searchName)) {

return product;

}

}

return null;

}

public SearchProduct binarySearch(SearchProduct[] products, String searchName) {

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = low + (high - low) / 2;

int comparison = searchName.compareToIgnoreCase(products[mid].getProductName());

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

high = mid - 1;

} else {

low = mid + 1;

}

}

return null;

}

}

**ECommerceSearch.java:**

import java.util.Arrays;

import java.util.Comparator;

class ECommerceSearch {

public static void main(String[] args) {

SearchProduct[] products = {

new SearchProduct("SP001", "Laptop", "Electronics"),

new SearchProduct("SP002", "Smartphone", "Electronics"),

new SearchProduct("SP003", "Desk Chair", "Furniture"),

new SearchProduct("SP004", "Coffee Maker", "Appliances"),

new SearchProduct("SP005", "Running Shoes", "Apparel")

};

ProductSearcher searcher = new ProductSearcher();

System.out.println("--- Linear Search Demonstration ---");

SearchProduct foundProduct = searcher.linearSearch(products, "Smartphone");

if (foundProduct != null) {

System.out.println("Found (Linear): " + foundProduct);

} else {

System.out.println("Product 'Smartphone' not found (Linear).");

}

foundProduct = searcher.linearSearch(products, "NonExistentItem");

if (foundProduct != null) {

System.out.println("Found (Linear): " + foundProduct);

} else {

System.out.println("Product 'NonExistentItem' not found (Linear).");

}

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

// For Binary Search, the array must be sorted by the search key (productName)

Arrays.sort(products, Comparator.comparing(SearchProduct::getProductName));

System.out.println("Products sorted by name for Binary Search:");

for(SearchProduct p : products) {

System.out.println(p.getProductName());

}

foundProduct = searcher.binarySearch(products, "Desk Chair");

if (foundProduct != null) {

System.out.println("Found (Binary): " + foundProduct);

} else {

System.out.println("Product 'Desk Chair' not found (Binary).");

}

foundProduct = searcher.binarySearch(products, "Laptop");

if (foundProduct != null) {

System.out.println("Found (Binary): " + foundProduct);

} else {

System.out.println("Product 'Laptop' not found (Binary).");

}

foundProduct = searcher.binarySearch(products, "NonExistentItem");

if (foundProduct != null) {

System.out.println("Found (Binary): " + foundProduct);

} else {

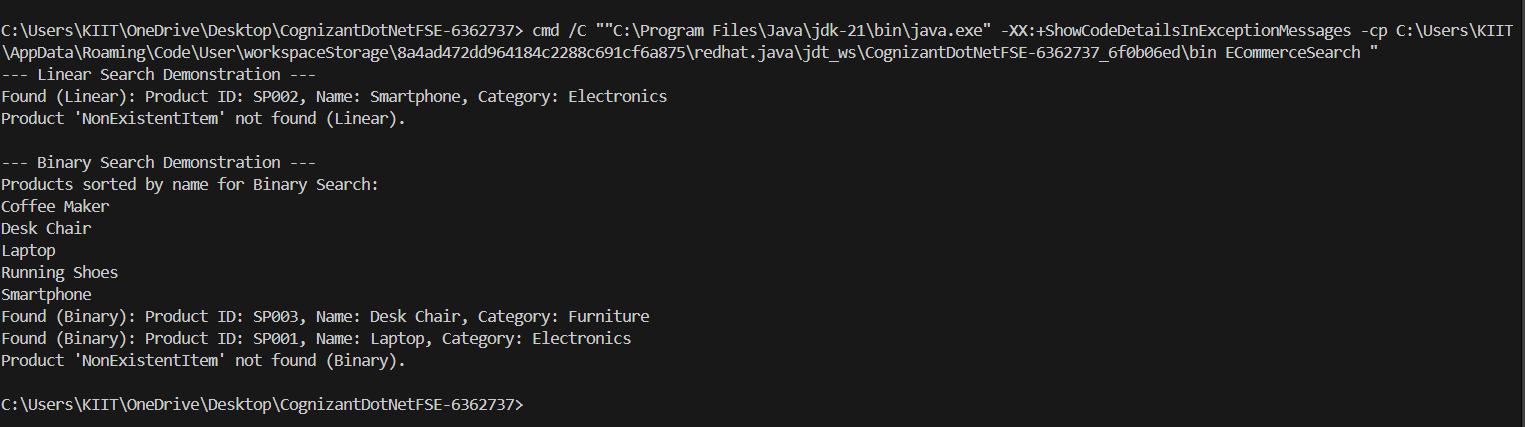
System.out.println("Product 'NonExistentItem' not found (Binary).");

}

}

}

**Output:**



**Analysis:**

**1. What is the time complexity of linear and binary search?**

* **Linear Search:** The time complexity is O(n) for the average and worst cases, meaning the search time grows linearly with the size of the list. The best case is O(1) if the item is at the beginning of the list.
* **Binary Search:** The time complexity is O(logn) for the average and worst cases, making it significantly faster for large lists as it halves the search space with each step. Its best case is also O(1) if the item is the middle element.

**2. Which algorithm is more suitable for my platform and why?**

* **Binary search** is more suitable for a large-scale platform like this one. The reason is that this platform processes vast amounts of structured and indexed (sorted) information. Binary search's logarithmic time complexity (O(logn)) is essential for quickly searching through large datasets to provide a fast and responsive user experience, whereas a linear search would be impractically slow.

**Exercise 3: Sorting Customer Orders**

**Steps:**

1. **Understand Sorting Algorithms:**

### 1. Bubble Sort

* **Definition:** A simple comparison-based algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. It continues until no swaps are needed in a pass, indicating the list is sorted.
* **Time Complexity:**
  + **Best Case:** O(n)
  + **Average Case:** O(n2)
  + **Worst Case:** O(n2)

### 2. Insertion Sort

* **Definition:** An algorithm that builds the final sorted array one item at a time. It iterates through the input elements, taking one element and inserting it into its correct position in the already sorted part of the array.
* **Time Complexity:**
  + **Best Case:** O(n)
  + **Average Case:** O(n2)
  + **Worst Case:** O(n2)

### 3. Quick Sort

* **Definition:** A highly efficient, divide-and-conquer algorithm that picks an element as a "pivot" and partitions the array around it. All elements smaller than the pivot are moved to its left, and all elements greater are moved to its right. This process is then recursively applied to the sub-arrays.
* **Time Complexity:**
  + **Best Case:** O(nlogn)
  + **Average Case:** O(nlogn)
  + **Worst Case:** O(n2)

### 4. Merge Sort

* **Definition:** A divide-and-conquer algorithm that recursively divides the unsorted list into single-element sub-lists and then repeatedly merges these sorted sub-lists to produce new sorted sub-lists until there is only one sorted list remaining.
* **Time Complexity:**
  + **Best Case:** O(nlogn)
  + **Average Case:** O(nlogn)
  + **Worst Case:** O(nlogn)

**Implementation:**

**Order.java:**

class Order {

private String orderId;

private String customerName;

private double totalPrice;

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

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public String getOrderId() {

return orderId;

}

public String getCustomerName() {

return customerName;

}

public double getTotalPrice() {

return totalPrice;

}

public String toString() {

return "Order ID: " + orderId + ", Customer: " + customerName +

", Total Price: $" + String.format("%.2f", totalPrice);

}

}

**OrderSorter.java:**

class OrderSorter {

public void bubbleSort(Order[] orders) {

int n = orders.length;

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

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

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

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

public void quickSort(Order[] orders) {

if (orders == null || orders.length == 0) {

return;

}

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

}

private void quickSort(Order[] orders, int low, int high) {

if (low < high) {

int pi = partition(orders, low, high);

quickSort(orders, low, pi - 1);

quickSort(orders, pi + 1, high);

}

}

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

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

int i = (low - 1);

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

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

i++;

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

Order temp = orders[i + 1];

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

orders[high] = temp;

return i + 1;

}

}

**SortingCustomerOrders.java:**

import java.util.Arrays;

class SortingCustomerOrders {

public static void main(String[] args) {

Order[] orders1 = {

new Order("OR005", "Bob", 150.75),

new Order("OR001", "Alice", 250.00),

new Order("OR003", "Charlie", 75.20),

new Order("OR002", "David", 300.50),

new Order("OR004", "Eve", 100.99)

};

Order[] orders2 = Arrays.copyOf(orders1, orders1.length); // Create a copy for Quick Sort

OrderSorter sorter = new OrderSorter();

System.out.println("--- Original Orders ---");

for (Order order : orders1) {

System.out.println(order);

}

System.out.println("\n--- Bubble Sort Orders by Total Price ---");

sorter.bubbleSort(orders1);

for (Order order : orders1) {

System.out.println(order);

}

System.out.println("\n--- Quick Sort Orders by Total Price ---");

sorter.quickSort(orders2);

for (Order order : orders2) {

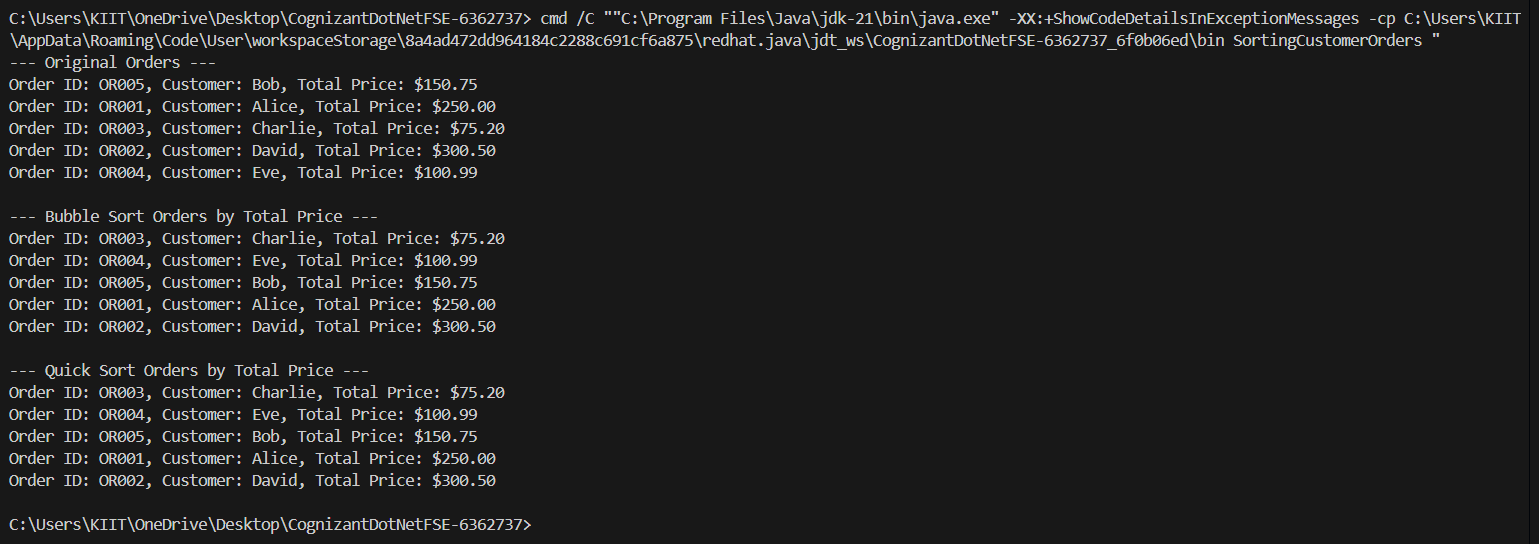
System.out.println(order);

}

}

}

Output:



1. **Analysis:**

* Bubble Sort consistently has a quadratic time complexity (O(n2)) for average and worst-case scenarios. Even its best case (O(n)) is only achieved when the array is already sorted, which is a very specific and often unlikely scenario.
* **Quick Sort** exhibits a significantly better average and best-case time complexity of O(nlogn). This logarithmic factor makes a huge difference for larger datasets. While its worst-case is also O(n2), this scenario is rare in practice with good pivot selection strategies.

### Why Quick Sort is Generally Preferred Over Bubble Sort

**Superior Average-Case Performance:** The primary reason is Quick Sort's average-case time complexity of O(nlogn) compared to Bubble Sort's O(n2).

**"Divide and Conquer" Efficiency:** Quick Sort's divide-and-conquer strategy inherently breaks down the problem into smaller, more manageable subproblems, which it solves recursively. This approach is generally more efficient than Bubble Sort's iterative, element-by-element comparison.

**In-Place Sorting (Memory Efficiency):** Both are in-place sorting algorithms (O(1) space for Bubble Sort, O(logn) for Quick Sort's recursion stack). However, given Quick Sort's speed, its minor additional space overhead for recursion is a small price to pay for the significant performance gains.

**Practicality and Optimization:** While Quick Sort has a theoretical worst-case of O(n2), modern implementations and common pivot selection strategies (like median-of-three or random pivot) make this worst case extremely rare in practice. Bubble Sort, on the other hand, will always perform O(n2) comparisons and swaps (unless optimized for already sorted arrays, but even then, it's inefficient for general cases).

**Exercise 4: Employee Management System**

**Steps:**

**Understand Array Representation:**

**Arrays in Memory:**

* Arrays are stored in contiguous memory locations
* Each element can be accessed using index calculation: base\_address + (index \* element\_size)
* **Advantages:**
  + Fast random access (O(1) time complexity)
  + Cache-friendly due to spatial locality
  + Memory efficient (no extra pointers needed)
  + Simple to implement and understand

**Implementation:**

**Employee.java:**

public class Employee {

private String employeeId;

private String name;

private String position;

private double salary;

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

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

public String getEmployeeId() {

return employeeId;

}

public String getName() {

return name;

}

public String getPosition() {

return position;

}

public void setPosition(String position) {

this.position = position;

}

public double getSalary() {

return salary;

}

public void setSalary(double salary) {

this.salary = salary;

}

public String toString() {

return "Employee ID: " + employeeId + ", Name: " + name + ", Position: " + position + ", Salary: $" + String.format("%.2f", salary);

}

}

**EmployeeRecordsManager.java:**

package exercise4;

import java.util.Arrays;

public class EmployeeRecordsManager {

private Employee[] employees;

private int count;

private static final int INITIAL\_CAPACITY = 10;

public EmployeeRecordsManager() {

this.employees = new Employee[INITIAL\_CAPACITY];

this.count = 0;

}

private void ensureCapacity() {

if (count == employees.length) {

employees = Arrays.copyOf(employees, employees.length \* 2);

}

}

public void addEmployee(Employee employee) {

ensureCapacity();

employees[count++] = employee;

}

public Employee searchEmployee(String employeeId) {

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

if (employees[i].getEmployeeId().equals(employeeId)) {

return employees[i];

}

}

return null;

}

public void traverseEmployees() {

if (count == 0) {

System.out.println("No employee records to display.");

return;

}

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

System.out.println(employees[i]);

}

}

public boolean deleteEmployee(String employeeId) {

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

if (employees[i].getEmployeeId().equals(employeeId)) {

// Shift elements to the left to fill the gap

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

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

}

employees[count - 1] = null; // Clear the last element

count--;

return true;

}

}

return false;

}

}

EmployeeManagementSystem.java:

package exercise4;

public class EmployeeManagementSystem {

public static void main(String[] args) {

EmployeeRecordsManager manager = new EmployeeRecordsManager();

System.out.println("--- Adding Employees ---");

manager.addEmployee(new Employee("E001", "Alice Smith", "Software Engineer", 75000.00));

manager.addEmployee(new Employee("E002", "Bob Johnson", "Project Manager", 90000.00));

manager.addEmployee(new Employee("E003", "Charlie Brown", "HR Specialist", 60000.00));

manager.traverseEmployees();

System.out.println("\n--- Searching for Employee E002 ---");

Employee foundEmployee = manager.searchEmployee("E002");

if (foundEmployee != null) {

System.out.println("Found: " + foundEmployee);

} else {

System.out.println("Employee E002 not found.");

}

System.out.println("\n--- Searching for non-existent Employee E004 ---");

foundEmployee = manager.searchEmployee("E004");

if (foundEmployee != null) {

System.out.println("Found: " + foundEmployee);

} else {

System.out.println("Employee E004 not found.");

}

System.out.println("\n--- Deleting Employee E002 ---");

boolean deleted = manager.deleteEmployee("E002");

System.out.println("Employee E002 deleted: " + deleted);

System.out.println("Employees after deletion:");

manager.traverseEmployees();

System.out.println("\n--- Adding more employees to test capacity ---");

for (int i = 4; i <= 12; i++) {

manager.addEmployee(new Employee("E00" + i, "New Employee " + i, "Associate", 50000.00 + i \* 100));

}

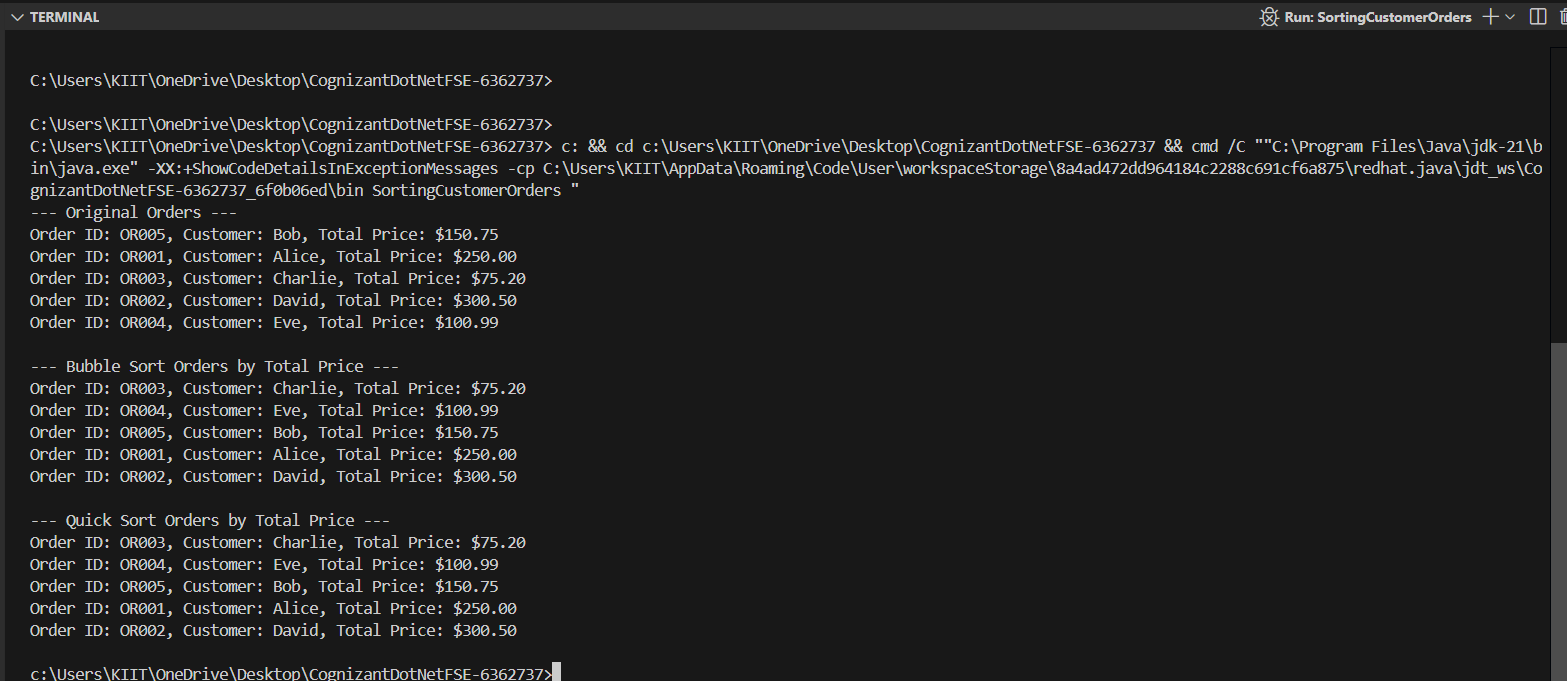
System.out.println("Employees after adding more:");

manager.traverseEmployees();

}

}

Output:



1. **Analysis:**

### Time Complexity of Array Operations

* **Add:**
  + **At End (Append):** O(1) amortized (if space is available); O(n) if resizing.
  + **At Beginning/Middle:** O(n) (requires shifting elements).
* **Search:**
  + **By Index:** O(1) (direct access).
  + **By Value (Unsorted):** O(n) (linear scan).
  + **By Value (Sorted - Binary Search):** O(logn).
* **Traverse:** O(n) (visit all n elements).
* **Delete:**
  + **From End:** O(1).
  + **From Beginning/Middle:** O(n) (requires shifting elements).

### Limitations of Arrays

1. **Fixed Size:** Static arrays have a fixed size defined at creation; resizing requires creating a new array and copying elements (O(n)).
2. **Inefficient Middle Operations:** Adding or deleting in the middle is slow (O(n)) due to the need to shift subsequent elements.
3. **Homogeneous Data (in some languages):** Typically store elements of a single data type.

### When to Use Arrays

1. **Fast Random Access:** When O(1) access by index is crucial.
2. **Known/Fixed Size:** When the number of elements is stable or determined beforehand.
3. **Memory Locality:** For performance benefits due to contiguous memory storage.
4. **Building Blocks:** As the underlying structure for other data structures (e.g., stacks, queues, hash tables, dynamic arrays).

**Exercise 5: Task Management System**

**Scenario:**

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

**Steps:**

1. **Understand Linked Lists:**

**Types of Linked Lists:**

1. **Singly Linked List:**
   1. Each node contains data and a pointer to the next node
   2. Traversal is unidirectional
   3. Memory efficient
2. **Doubly Linked List:**
   1. Each node has pointers to both next and previous nodes
   2. Bidirectional traversal
   3. More memory overhead but faster deletion

**Advantages:**

* Dynamic size
* Efficient insertion/deletion at beginning
* No memory waste

1. **Setup:**

**Task.java:**

package exercise5;

public class Task {

private String taskId;

private String taskName;

private String status; // e.g., "Pending", "In Progress", "Completed"

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

public String getTaskId() {

return taskId;

}

public String getTaskName() {

return taskName;

}

public String getStatus() {

return status;

}

public void setStatus(String status) {

this.status = status;

}

public String toString() {

return "Task ID: " + taskId + ", Name: " + taskName + ", Status: " + status;

}

}

1. **Implementation:**

**TaskNode.java:**

package exercise5;

public class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

**TaskListManager.java:**

package exercise5;

public class TaskListManager {

private TaskNode head;

public TaskListManager() {

this.head = null;

}

public void addTask(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;

}

}

public Task searchTask(String taskId) {

TaskNode current = head;

while (current != null) {

if (current.task.getTaskId().equals(taskId)) {

return current.task;

}

current = current.next;

}

return null;

}

public void traverseTasks() {

if (head == null) {

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

return;

}

TaskNode current = head;

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

public boolean deleteTask(String taskId) {

if (head == null) {

return false;

}

if (head.task.getTaskId().equals(taskId)) {

head = head.next;

return true;

}

TaskNode current = head;

while (current.next != null && !current.next.task.getTaskId().equals(taskId)) {

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

return true;

}

return false;

}

}

**TaskManagementSystem.java:**

package exercise5;

public class TaskManagementSystem {

public static void main(String[] args) {

TaskListManager manager = new TaskListManager();

System.out.println("--- Adding Tasks ---");

manager.addTask(new Task("T001", "Design Database Schema", "Pending"));

manager.addTask(new Task("T002", "Implement User Authentication", "In Progress"));

manager.addTask(new Task("T003", "Write API Documentation", "Pending"));

manager.traverseTasks();

System.out.println("\n--- Searching for Task T002 ---");

Task foundTask = manager.searchTask("T002");

if (foundTask != null) {

System.out.println("Found: " + foundTask);

} else {

System.out.println("Task T002 not found.");

}

System.out.println("\n--- Searching for non-existent Task T004 ---");

foundTask = manager.searchTask("T004");

if (foundTask != null) {

System.out.println("Found: " + foundTask);

} else {

System.out.println("Task T004 not found.");

}

System.out.println("\n--- Deleting Task T002 ---");

boolean deleted = manager.deleteTask("T002");

System.out.println("Task T002 deleted: " + deleted);

System.out.println("Tasks after deletion:");

manager.traverseTasks();

System.out.println("\n--- Adding another task ---");

manager.addTask(new Task("T004", "Deploy to Production", "Pending"));

manager.traverseTasks();

System.out.println("\n--- Deleting first task T001 ---");

manager.deleteTask("T001");

System.out.println("Tasks after deleting T001:");

manager.traverseTasks();

System.out.println("\n--- Deleting last task T004 ---");

manager.deleteTask("T004");

System.out.println("Tasks after deleting T004:");

manager.traverseTasks();

System.out.println("\n--- Deleting remaining task T003 ---");

manager.deleteTask("T003");

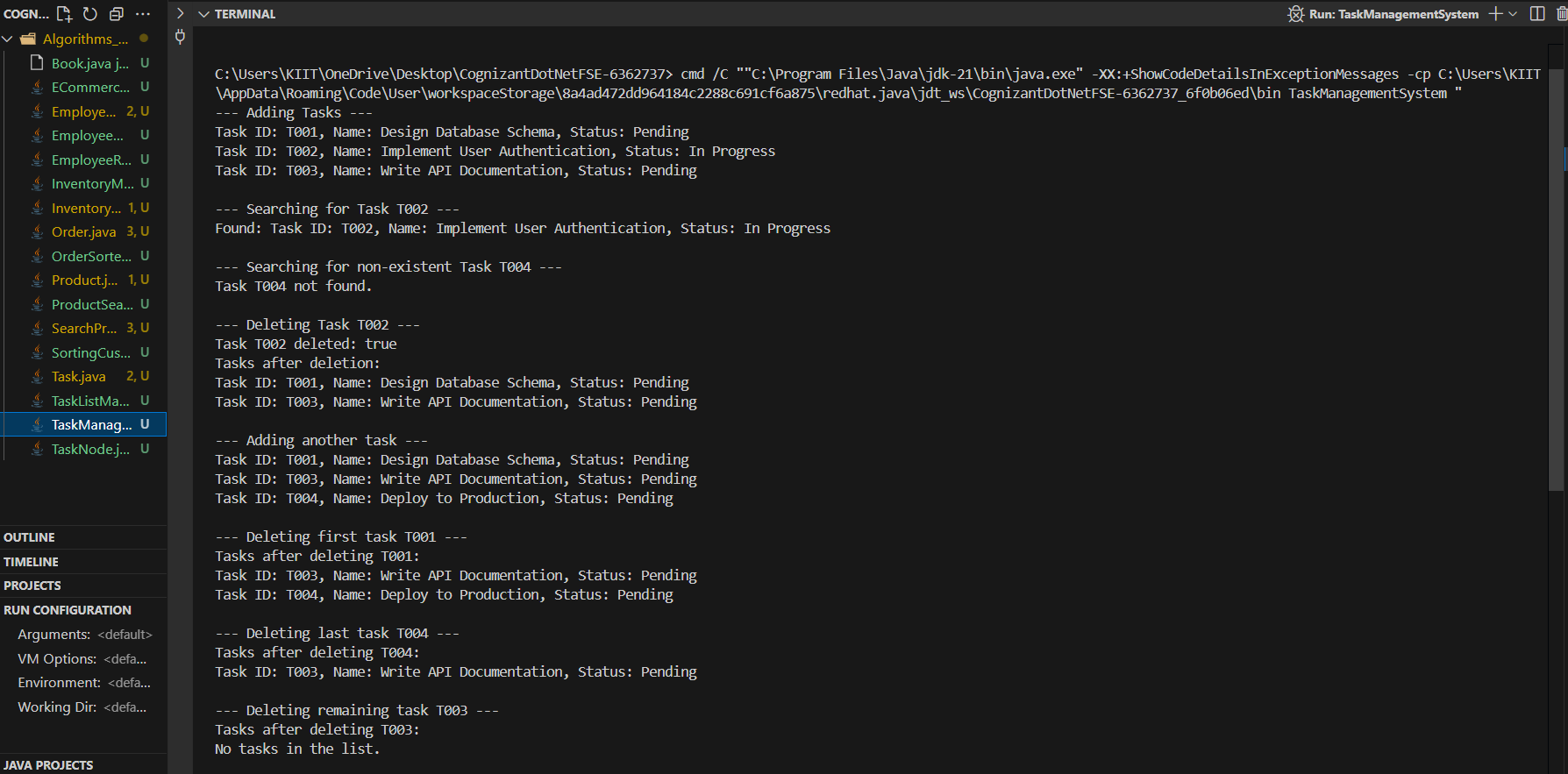
System.out.println("Tasks after deleting T003:");

manager.traverseTasks(); // Should show "No tasks in the list."

}

}

Output:



1. **Analysis:**

**Time Complexity Analysis:**

* **Add:** O(n) - need to traverse to end
* **Search:** O(n) - linear search
* **Traverse:** O(n) - visit all nodes
* **Delete:** O(n) - need to find the node

**Advantages over Arrays:**

* Dynamic size
* Efficient insertion at beginning O(1)
* No memory reallocation needed
* Memory allocated as needed

**Exercise 6: Library Management System**

**Scenario:**

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

**Steps:**

1. **Understand Search Algorithms:**

**Linear Search:**

* Sequentially checks each element
* Time Complexity: O(n)
* Works on unsorted data
* Simple to implement

**Binary Search:**

* Divides search space in half each iteration
* Time Complexity: O(log n)
* Requires sorted data
* More efficient for large datasets

1. **Setup:**

**Book.java**

package exercise6;

public class Book {

private String bookId;

private String title;

private String author;

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

this.bookId = bookId;

this.title = title;

this.author = author;

}

/\*

Returns the book ID.

return The book ID.

\*/

public String getBookId() {

return bookId;

}

/\*

Returns the title of the book.

return The title.

\*/

public String getTitle() {

return title;

}

/\*

Returns the author of the book.

return The author.

\*/

public String getAuthor() {

return author;

}

/\*

Returns a string representation of the Book.

return A string containing book details.

/

public String toString() {

return "Book ID: " + bookId + ", Title: \"" + title + "\", Author: " + author;

}

}

1. **Implementation:**

**LibrarySearcher.java:**

package exercise6;

public class LibrarySearcher {

public Book linearSearchByTitle(Book[] books, String searchTitle) {

for (Book book : books) {

if (book.getTitle().equalsIgnoreCase(searchTitle)) {

return book;

}

}

return null;

}

public Book binarySearchByTitle(Book[] books, String searchTitle) {

int low = 0;

int high = books.length - 1;

while (low <= high) {

int mid = low + (high - low) / 2;

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

if (comparison == 0) {

return books[mid];

} else if (comparison < 0) {

high = mid - 1;

} else {

low = mid + 1;

}

}

return null;

}

}

**LibraryManagementSystem.java:**

import java.util.Arrays;

import java.util.Comparator;

public class LibraryManagementSystem {

public static void main(String[] args) {

Book[] libraryBooks = {

new Book("B001", "The Great Gatsby", "F. Scott Fitzgerald"),

new Book("B002", "1984", "George Orwell"),

new Book("B003", "To Kill a Mockingbird", "Harper Lee"),

new Book("B004", "Pride and Prejudice", "Jane Austen"),

new Book("B005", "The Catcher in the Rye", "J.D. Salinger")

};

LibrarySearcher searcher = new LibrarySearcher();

System.out.println("--- Linear Search by Title Demonstration ---");

Book foundBook = searcher.linearSearchByTitle(libraryBooks, "1984");

if (foundBook != null) {

System.out.println("Found (Linear): " + foundBook);

} else {

System.out.println("Book '1984' not found (Linear).");

}

foundBook = searcher.linearSearchByTitle(libraryBooks, "Moby Dick");

if (foundBook != null) {

System.out.println("Found (Linear): " + foundBook);

} else {

System.out.println("Book 'Moby Dick' not found (Linear).");

}

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

// For Binary Search, the array must be sorted by title

Arrays.sort(libraryBooks, Comparator.comparing(Book::getTitle));

System.out.println("Books sorted by Title for Binary Search:");

for (Book book : libraryBooks) {

System.out.println(" " + book.getTitle());

}

foundBook = searcher.binarySearchByTitle(libraryBooks, "Pride and Prejudice");

if (foundBook != null) {

System.out.println("Found (Binary): " + foundBook);

} else {

System.out.println("Book 'Pride and Prejudice' not found (Binary).");

}

foundBook = searcher.binarySearchByTitle(libraryBooks, "The Great Gatsby");

if (foundBook != null) {

System.out.println("Found (Binary): " + foundBook);

} else {

System.out.println("Book 'The Great Gatsby' not found (Binary).");

}

foundBook = searcher.binarySearchByTitle(libraryBooks, "War and Peace");

if (foundBook != null) {

System.out.println("Found (Binary): " + foundBook);

} else {

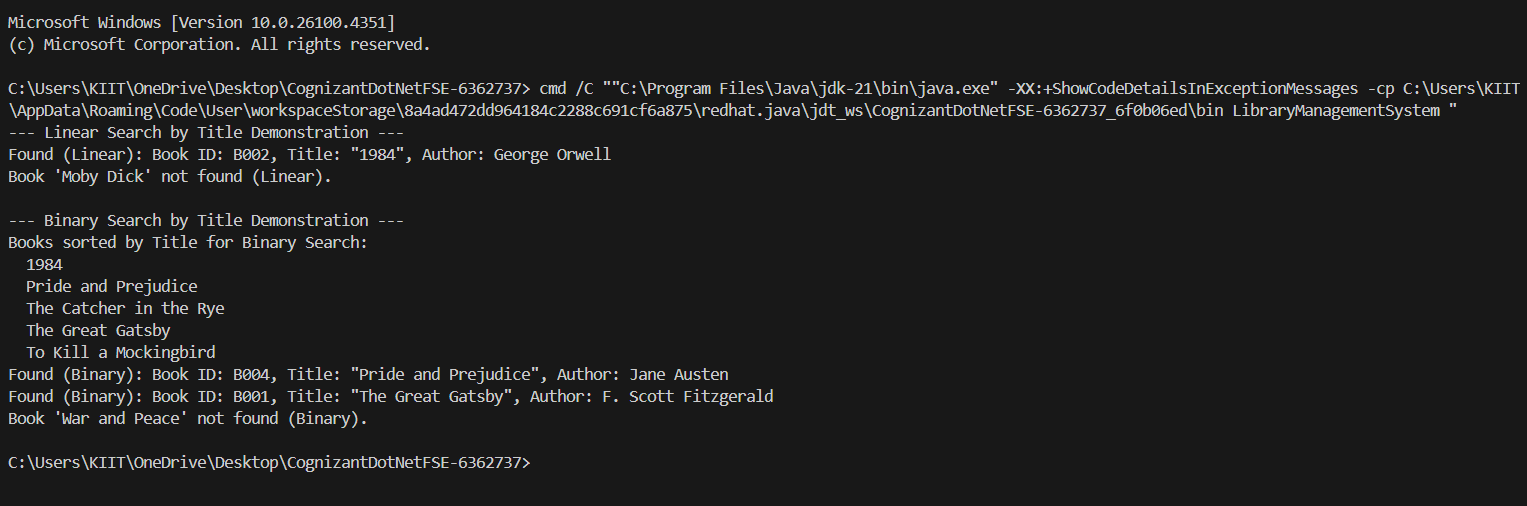
System.out.println("Book 'War and Peace' not found (Binary).");

}

}

}

Output:



1. **Analysis:**

**Time Complexity Comparison:**

* **Linear Search:** O(n) - worst case checks all elements
* **Binary Search:** O(log n) - eliminates half the search space each iteration

**When to Use:**

* **Linear Search:** Small datasets, unsorted data, simple implementation
* **Binary Search:** Large datasets, sorted data, performance critical

**Space Complexity:**

* Both algorithms use O(1) extra space

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**

FinancialCalculatorUtils.java

1. **Implementation:**

public class FinancialCalculatorUtils {

Validates input parameters for financial calculations.

principal The principal amount

rate The interest rate

periods The number of periods

throws IllegalArgumentException if any parameter is invalid

\*/

public static void validateFinancialInputs(double principal, double rate, int periods) {

if (principal < 0) {

throw new IllegalArgumentException("Principal amount cannot be negative.");

}

if (rate < -1.0) {

throw new IllegalArgumentException("Interest rate cannot be less than -100%.");

}

if (periods < 0) {

throw new IllegalArgumentException("Number of periods cannot be negative.");

}

}

/\*

Calculates the compound annual growth rate (CAGR) between two values.

\*/

public static double calculateCAGR(double beginningValue, double endingValue, int periods) {

validateFinancialInputs(beginningValue, 0, periods);

if (endingValue <= 0) {

throw new IllegalArgumentException("Ending value must be positive.");

}

if (periods == 0) {

return 0.0;

}

return Math.pow(endingValue / beginningValue, 1.0 / periods) - 1.0;

}

/\*

Calculates the number of periods required to reach a target amount.

\*/

public static double calculatePeriodsRequired(double presentValue, double futureValue, double interestRate) {

validateFinancialInputs(presentValue, interestRate, 0);

if (futureValue <= presentValue) {

throw new IllegalArgumentException("Future value must be greater than present value for positive growth.");

}

if (interestRate <= 0) {

throw new IllegalArgumentException("Interest rate must be positive for growth calculations.");

}

return Math.log(futureValue / presentValue) / Math.log(1 + interestRate);

}

/\*

Calculates the effective annual rate given a nominal rate and compounding frequency.

nominalRate The nominal annual interest rate

compoundingFrequency The number of compounding periods per year

return The effective annual rate

\*/

public static double calculateEffectiveAnnualRate(double nominalRate, int compoundingFrequency) {

if (nominalRate < 0) {

throw new IllegalArgumentException("Nominal rate cannot be negative.");

}

if (compoundingFrequency <= 0) {

throw new IllegalArgumentException("Compounding frequency must be positive.");

}

return Math.pow(1 + nominalRate / compoundingFrequency, compoundingFrequency) - 1;

}

public static String formatCurrency(double amount) {

return String.format("$%,.2f", amount);

}

public static String formatPercentage(double rate, int decimals) {

String format = "%." + decimals + "f%%";

return String.format(format, rate \* 100);

}

public static double calculateTotalInterest(double principal, double futureValue) {

return futureValue - principal;

}

public static double calculateBreakEvenYears(double initialInvestment, double annualReturn, double annualCosts) {

if (annualReturn <= 0) {

throw new IllegalArgumentException("Annual return must be positive to break even.");

}

if (annualCosts < 0) {

throw new IllegalArgumentException("Annual costs cannot be negative.");

}

// Simplified break-even calculation

if (annualCosts == 0) {

return 0; // No costs, immediate break-even

}

return initialInvestment / (initialInvestment \* annualReturn - annualCosts);

}

public static void runUtilityTests() {

System.out.println("=== FINANCIAL CALCULATOR UTILITIES TEST ===\n");

try {

// Test CAGR calculation

double cagr = calculateCAGR(1000, 1500, 5);

System.out.printf("CAGR Test: From $1000 to $1500 in 5 years = %s\n",

formatPercentage(cagr, 2));

// Test periods required calculation

double periodsNeeded = calculatePeriodsRequired(1000, 2000, 0.07);

System.out.printf("Periods Required: To double $1000 at 7%% = %.1f years\n", periodsNeeded);

// Test effective annual rate

double ear = calculateEffectiveAnnualRate(0.06, 12);

System.out.printf("Effective Annual Rate: 6%% compounded monthly = %s\n",

formatPercentage(ear, 3));

// Test formatting functions

System.out.printf("Currency Formatting: %s\n", formatCurrency(12345.678));

System.out.printf("Percentage Formatting: %s\n", formatPercentage(0.0825, 2));

// Test interest calculation

double interest = calculateTotalInterest(1000, 1628.89);

System.out.printf("Total Interest: %s\n", formatCurrency(interest));

// Test break-even calculation

double breakEven = calculateBreakEvenYears(10000, 0.08, 500);

System.out.printf("Break-even: %.1f years for $10k investment at 8%% with $500 annual costs\n",

breakEven);

System.out.println("✓ All utility tests completed successfully!");

} catch (Exception e) {

System.err.println("✗ Error in utility tests: " + e.getMessage());

}

}

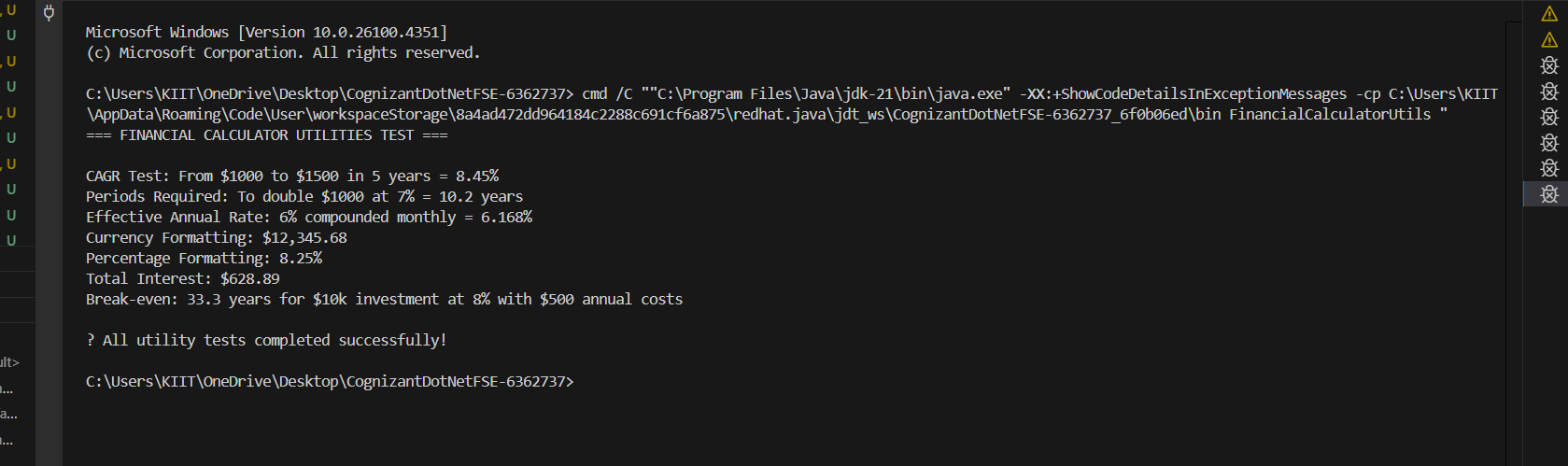
public static void main(String[] args) {

runUtilityTests();

}

}

Output:



1. **Analysis:**

## Time Complexity of the Recursive Algorithm

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

* The function makes exactly n recursive calls before reaching the base case
* Each recursive call performs constant-time operations (one multiplication)
* The total number of operations grows linearly with the number of periods

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

* Each recursive call creates a new stack frame
* The maximum stack depth equals the number of periods
* This creates risk of stack overflow for large period values

## Optimizations to Avoid Excessive Computation

### 1. **Convert to Iterative Approach**

* Eliminates the function call overhead and stack space usage
* Reduces space complexity from O(n) to O(1)
* Removes stack overflow risk for large periods

### 2. **Use Direct Mathematical Formula**

* Apply the compound interest formula: FV = PV × (1 + r)^n
* Reduces time complexity from O(n) to O(1)
* Most efficient for single calculations

### 3. **Implement Tail Recursion**

* Restructure the recursion so the recursive call is the last operation
* Enables compiler optimization to convert recursion to iteration
* Maintains recursive structure while improving performance

### 4. **Add Memoization**

* Cache previously computed results to avoid redundant calculations
* Beneficial when performing multiple related calculations
* Trades memory for computation time in scenarios with overlapping subproblems