**Data Structures and Algorithms Assignment Solutions**

**Exercise 1: Inventory Management System  
  
Scenario:**

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

**Steps:**

1. **Understand the Problem:**
   * Explain why data structures and algorithms are essential in handling large inventories.
   * Discuss the types of data structures suitable for this problem.
2. **Setup:**
   * Create a new project for the inventory management system.
3. **Implementation:**
   * Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.
   * Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
   * Implement methods to add, update, and delete products from the inventory.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
   * Discuss how you can optimize these operations.

**Solution**

**File: Product.java**

public class Product {

private int id;

private String name;

private int qty;

private double price;

public Product(int id, String name, int qty, double price) {

this.id = id;

this.name = name;

this.qty = qty;

this.price = price;

}

public int getId() { return id; }

public String getName() { return name; }

public int getQty() { return qty; }

public double getPrice() { return price; }

public void setQty(int qty) { this.qty = qty; }

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

public String toString() {

return "Product[ID=" + id + ", Name=" + name + ", Qty=" + qty + ", Price=$" + price + "]";

}

}

**File: Inventory.java**

import java.util.\*;

public class Inventory {

private HashMap<Integer, Product> products;

public Inventory() {

products = new HashMap<>();

}

public void add(Product p) {

products.put(p.getId(), p);

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

}

public void update(int id, int newQty, double newPrice) {

Product p = products.get(id);

if (p != null) {

p.setQty(newQty);

p.setPrice(newPrice);

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

} else {

System.out.println("Product not found!");

}

}

public void delete(int id) {

Product removed = products.remove(id);

if (removed != null) {

System.out.println("Deleted: " + removed);

} else {

System.out.println("Product not found!");

}

}

public void display() {

System.out.println("\n=== Current Inventory ===");

for (Product p : products.values()) {

System.out.println(p);

}

}

}

**File: InventoryTest.java**

public class InventoryTest {

public static void main(String[] args) {

Inventory inv = new Inventory();

inv.add(new Product(1, "Laptop", 10, 999.99));

inv.add(new Product(2, "Mouse", 50, 25.99));

inv.add(new Product(3, "Keyboard", 30, 79.99));

inv.update(1, 8, 949.99);

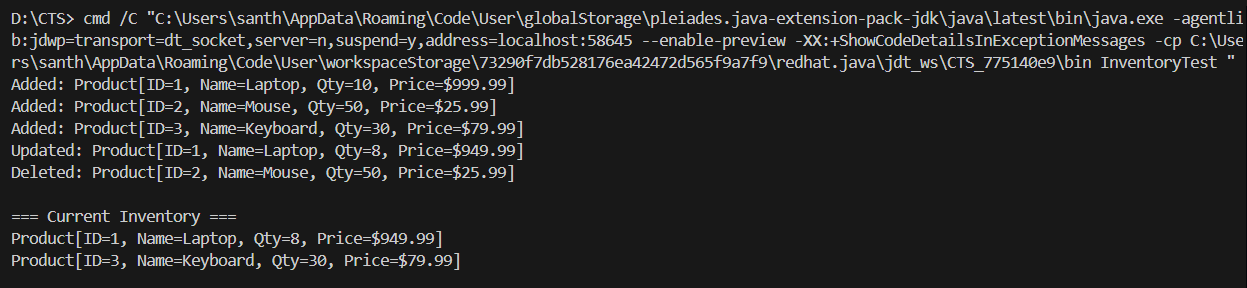
inv.delete(2);

inv.display();

}

}

**Output:**



**Time Complexity Analysis**

* **Add Operation**: O(1) - HashMap insertion
* **Update Operation**: O(1) - HashMap lookup and modification
* **Delete Operation**: O(1) - HashMap removal

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

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

**Steps:**

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

**Solution**

**File: SearchProduct.java**

public class SearchProduct {

private int id;

private String name;

private String category;

public SearchProduct(int id, String name, String category) {

this.id = id;

this.name = name;

this.category = category;

}

public int getId() { return id; }

public String getName() { return name; }

public String getCategory() { return category; }

public String toString() {

return "Product[ID=" + id + ", Name=" + name + ", Category=" + category + "]";

}

}

**File: SearchAlgorithms.java**

public class SearchAlgorithms {

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

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

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

return i;

}

}

return -1;

}

public static int binarySearch(SearchProduct[] products, int targetId) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

if (products[mid].getId() == targetId) {

return mid;

}

if (products[mid].getId() < targetId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1;

}

}

**File: SearchTest.java**

public class SearchTest {

public static void main(String[] args) {

SearchProduct[] products = {

new SearchProduct(1, "Phone", "Electronics"),

new SearchProduct(3, "Tablet", "Electronics"),

new SearchProduct(5, "Watch", "Accessories"),

new SearchProduct(7, "Headphones", "Audio")

};

int searchId = 5;

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

int binaryResult = SearchAlgorithms.binarySearch(products, searchId);

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

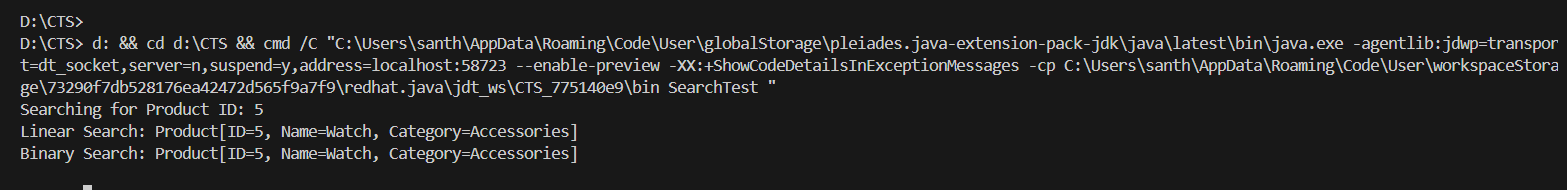
System.out.println("Linear Search: " + (linearResult != -1 ? products[linearResult] : "Not found"));

System.out.println("Binary Search: " + (binaryResult != -1 ? products[binaryResult] : "Not found"));

}

}

**Output:**



**Algorithm Comparison**

* **Linear Search**: O(n) - Better for unsorted data
* **Binary Search**: O(log n) - Better for sorted data, much faster for large datasets

**Exercise 3: Sorting Customer Orders**

**Scenario:**

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

**Steps:**

1. **Understand Sorting Algorithms:**
   * Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
2. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   * Discuss why Quick Sort is generally preferred over Bubble Sort.

**Solution**

**File: Order.java**

public class Order {

private int id;

private String customer;

private double total;

public Order(int id, String customer, double total) {

this.id = id;

this.customer = customer;

this.total = total;

}

public int getId() { return id; }

public String getCustomer() { return customer; }

public double getTotal() { return total; }

public String toString() {

return "Order[ID=" + id + ", Customer=" + customer + ", Total=$" + total + "]";

}

}

**File: SortingAlgorithms.java**

public class SortingAlgorithms {

public static 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].getTotal() > orders[j + 1].getTotal()) {

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

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

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

int i = (low - 1);

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

if (orders[j].getTotal() <= 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;

}

}

**File: SortingTest.java**

public class SortingTest {

public static void main(String[] args) {

Order[] orders1 = {

new Order(1, "John", 250.50),

new Order(2, "Alice", 125.75),

new Order(3, "Bob", 300.00),

new Order(4, "Carol", 180.25)

};

Order[] orders2 = {

new Order(1, "John", 250.50),

new Order(2, "Alice", 125.75),

new Order(3, "Bob", 300.00),

new Order(4, "Carol", 180.25)

};

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

for (Order o : orders1) {

System.out.println(o);

}

SortingAlgorithms.bubbleSort(orders1);

System.out.println("\nBubble Sort Result:");

for (Order o : orders1) {

System.out.println(o);

}

SortingAlgorithms.quickSort(orders2, 0, orders2.length - 1);

System.out.println("\nQuick Sort Result:");

for (Order o : orders2) {

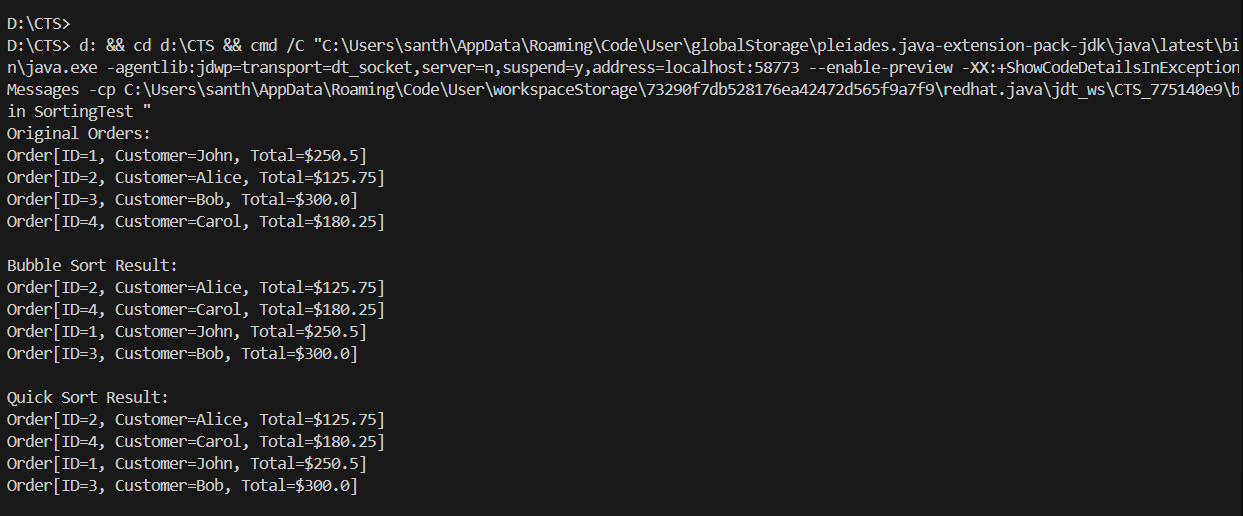
System.out.println(o);

}

}

}

**Output:**



**Why Quick Sort is Preferred**

* **Better Time Complexity**: O(n log n) vs O(n²)
* **In-place Sorting**: Uses less memory
* **Cache Efficient**: Better performance on modern processors

**Exercise 4: Employee Management System**

**Scenario:**

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

**Steps:**

1. **Understand Array Representation:**
   * Explain how arrays are represented in memory and their advantages.
2. **Setup:**
   * Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
3. **Implementation:**
   * Use an array to store employee records.
   * Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).
   * Discuss the limitations of arrays and when to use them.

**Solution**

**File: Employee.java**

public class Employee {

private int id;

private String name;

private String position;

private double salary;

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

this.id = id;

this.name = name;

this.position = position;

this.salary = salary;

}

public int getId() { return id; }

public String getName() { return name; }

public String getPosition() { return position; }

public double getSalary() { return salary; }

public String toString() {

return "Employee[ID=" + id + ", Name=" + name + ", Position=" + position + ", Salary=$" + salary + "]";

}

}

**File: EmployeeManager.java**

public class EmployeeManager {

private Employee[] employees;

private int count;

private int capacity;

public EmployeeManager(int capacity) {

this.capacity = capacity;

this.employees = new Employee[capacity];

this.count = 0;

}

public void add(Employee emp) {

if (count < capacity) {

employees[count] = emp;

count++;

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

} else {

System.out.println("Array is full!");

}

}

public Employee search(int id) {

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

if (employees[i].getId() == id) {

return employees[i];

}

}

return null;

}

public void traverse() {

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

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

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

}

}

public void delete(int id) {

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

if (employees[i].getId() == id) {

System.out.println("Deleted: " + employees[i]);

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

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

}

count--;

return;

}

}

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

}

}

**File: EmployeeTest.java**

public class EmployeeTest {

public static void main(String[] args) {

EmployeeManager mgr = new EmployeeManager(10);

mgr.add(new Employee(101, "John", "Developer", 75000));

mgr.add(new Employee(102, "Jane", "Manager", 85000));

mgr.add(new Employee(103, "Bob", "Analyst", 65000));

Employee found = mgr.search(102);

System.out.println("Search for ID 102: " + found);

mgr.traverse();

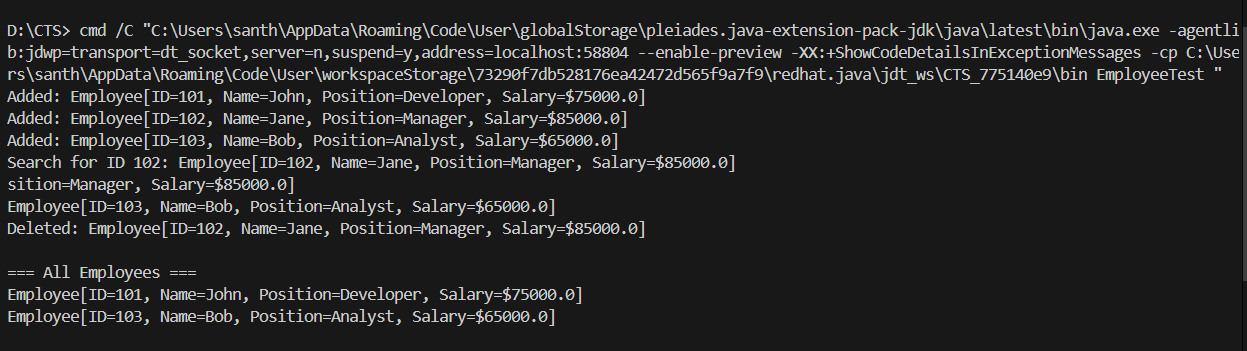
mgr.delete(102);

mgr.traverse();

}

}

**Output:**



**Time Complexity Analysis**

* **Add**: O(1) - Insert at end
* **Search**: O(n) - Linear search required
* **Traverse**: O(n) - Visit all elements
* **Delete**: O(n) - Find and shift elements

**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:**
   * Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
2. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
3. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.
4. **Analysis:**
   * Analyze the time complexity of each operation.
   * Discuss the advantages of linked lists over arrays for dynamic data.

**Solution**

**File: Task.java**

public class Task {

private int id;

private String name;

private String status;

public Task(int id, String name, String status) {

this.id = id;

this.name = name;

this.status = status;

}

public int getId() { return id; }

public String getName() { return name; }

public String getStatus() { return status; }

public String toString() {

return "Task[ID=" + id + ", Name=" + name + ", Status=" + status + "]";

}

}

**File: TaskNode.java**

public class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

**File: TaskList.java**

public class TaskList {

private TaskNode head;

public TaskList() {

this.head = null;

}

public void add(Task task) {

TaskNode newNode = new TaskNode(task);

newNode.next = head;

head = newNode;

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

}

public Task search(int id) {

TaskNode current = head;

while (current != null) {

if (current.task.getId() == id) {

return current.task;

}

current = current.next;

}

return null;

}

public void traverse() {

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

TaskNode current = head;

while (current != null) {

System.out.println(current.task);

current = current.next;

}

}

public void delete(int id) {

if (head == null) {

System.out.println("Task list is empty!");

return;

}

if (head.task.getId() == id) {

System.out.println("Deleted: " + head.task);

head = head.next;

return;

}

TaskNode current = head;

while (current.next != null) {

if (current.next.task.getId() == id) {

System.out.println("Deleted: " + current.next.task);

current.next = current.next.next;

return;

}

current = current.next;

}

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

}

}

**File: TaskTest.java**

public class TaskTest {

public static void main(String[] args) {

TaskList taskList = new TaskList();

taskList.add(new Task(1, "Design UI", "InProgress"));

taskList.add(new Task(2, "Write Tests", "Pending"));

taskList.add(new Task(3, "Code Review", "Completed"));

Task found = taskList.search(2);

System.out.println("Search for Task ID 2: " + found);

taskList.traverse();

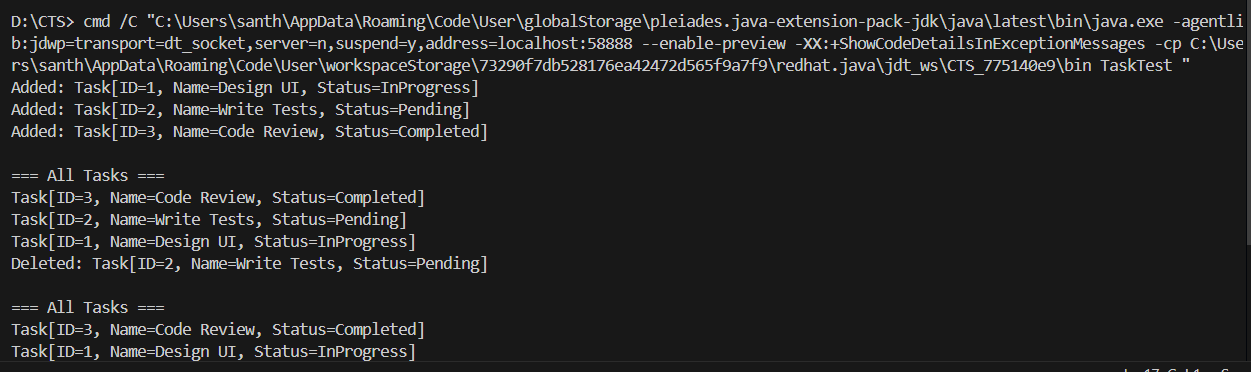
taskList.delete(2);

taskList.traverse();

}

}

**Output:**



**Advantages over Arrays:**

* Dynamic size
* Efficient insertion/deletion at beginning

**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:**
   * Explain linear search and binary search algorithms.
2. **Setup:**
   * Create a class **Book** with attributes like **bookId**, **title**, and **author**.
3. **Implementation:**
   * Implement linear search to find books by title.
   * Implement binary search to find books by title (assuming the list is sorted).
4. **Analysis:**
   * Compare the time complexity of linear and binary search.
   * Discuss when to use each algorithm based on the data set size and order.

**Solution**

**File: Book.java**

public class Book {

private int id;

private String title;

private String author;

public Book(int id, String title, String author) {

this.id = id;

this.title = title;

this.author = author;

}

public int getId() { return id; }

public String getTitle() { return title; }

public String getAuthor() { return author; }

public String toString() {

return "Book[ID=" + id + ", Title=" + title + ", Author=" + author + "]";

}

}

**File: LibrarySearch.java**

public class LibrarySearch {

public static Book linearSearch(Book[] books, String title) {

for (Book book : books) {

if (book.getTitle().equals(title)) {

return book;

}

}

return null;

}

public static Book binarySearch(Book[] books, String title) {

int left = 0;

int right = books.length - 1;

while (left <= right) {

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

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

if (comparison == 0) {

return books[mid];

}

if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

**File: LibraryTest.java**

public class LibraryTest {

public static void main(String[] args) {

Book[] books = {

new Book(1, "Clean Code", "Robert Martin"),

new Book(2, "Design Patterns", "Gang of Four"),

new Book(3, "Effective Java", "Joshua Bloch"),

new Book(4, "Java Concurrency", "Brian Goetz")

};

String searchTitle = "Clean Code";

Book linearResult = LibrarySearch.linearSearch(books, searchTitle);

Book binaryResult = LibrarySearch.binarySearch(books, searchTitle);

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

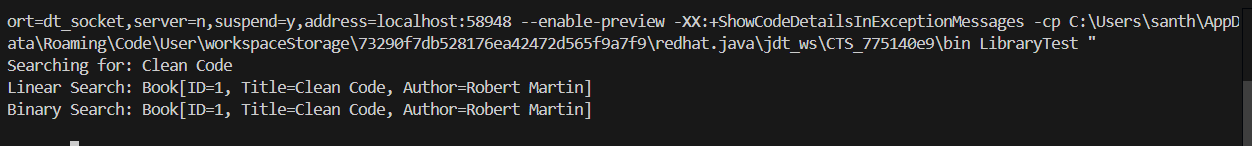
System.out.println("Linear Search: " + linearResult);

System.out.println("Binary Search: " + binaryResult);

}

}

**Output:**



**When to Use Each Algorithm**

* **Linear Search**: Small datasets, unsorted data
* **Binary Search**: Large datasets, sorted data (1000x faster for 1M records)

**Exercise 7: Financial Forecasting  
  
Scenario:**

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

**Steps:**

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

**Solution**

**File: FinancialForecasting.java**

public class FinancialForecasting {

public static double calculateFutureValue(double presentValue, double growthRate, int years) {

if (years == 0) {

return presentValue;

}

return calculateFutureValue(presentValue \* (1 + growthRate), growthRate, years - 1);

}

public static double calculateFutureValueIterative(double presentValue, double growthRate, int years) {

double futureValue = presentValue;

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

futureValue \*= (1 + growthRate);

}

return futureValue;

}

}

**File: ForecastingTest.java**

public class ForecastingTest {

public static void main(String[] args) {

double presentValue = 1000.0;

double growthRate = 0.05;

int years = 10;

double recursiveResult = FinancialForecasting.calculateFutureValue(presentValue, growthRate, years);

double iterativeResult = FinancialForecasting.calculateFutureValueIterative(presentValue, growthRate, years);

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

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

System.out.println("Years: " + years);

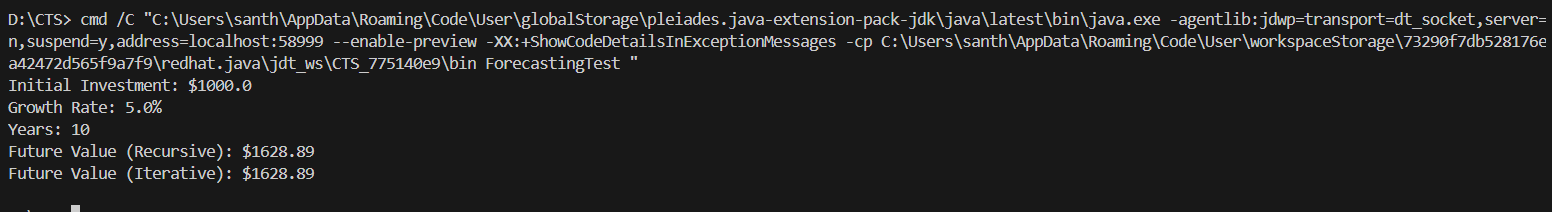
System.out.println("Future Value (Recursive): $" + String.format("%.2f", recursiveResult));

System.out.println("Future Value (Iterative): $" + String.format("%.2f", iterativeResult));

}

}

**Output:**



**Optimization Strategies**

1. **Iterative Approach**: Eliminates function call overhead
2. **Memoization**: Store computed results to avoid recomputation
3. **Mathematical Formula**: Use compound interest formula for O(1) solution