

## Exercise 1: Inventory Management System

1) Explain why data structures and algorithms are essential in handling large inventories.

- **Fast Access:** Quickly find products using IDs (e.g., with HashMap).
- **Efficient Updates:** Easily add, update, or delete items without slowing down.
- **Scalability:** Handle thousands of products smoothly as the system grows.
- **Memory Friendly:** Store data efficiently to save space.
- **Real-Time Performance:** Support quick actions needed for order processing and stock tracking.
- **Better Search & Sort:** Quickly filter or sort products using efficient algorithms.

2) Suitable Data Structures for This Problem:

- **ArrayList:**  
Good for **maintaining order** in small inventories, but **slow for searching or updating** by product ID ( $O(n)$ ).
- **LinkedList:**  
Allows **easy insertion/deletion** at ends, but **slow search** and **higher memory usage** make it less ideal for inventory.
- **HashMap (Best Choice):**  
Perfect for **large inventories** where you need to **add, update, delete, or search by productId quickly**.
  - **Operations:  $O(1)$**  on average.
  - Stores data as productId → Product pairs.
- **TreeMap:**  
Useful if you need **sorted order by productId**, but **slower ( $O(\log n)$ )** than HashMap.

### Code:

```
import java.util.HashMap;

public class Product {

    private String productId;

    private String productName;

    private int quantity;

    private double price;
```

```

public Product(String productId, String productName, int quantity, double price) {
    this.productId = productId;
    this.productName = productName;
    this.quantity = quantity;
    this.price = price;
}

// Getters and setters
public String getProductId() {
    return productId;
}

public String getProductName() {
    return productName;
}

public int getQuantity() {
    return quantity;
}

public double getPrice() {
    return price;
}

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

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

@Override
public String toString() {
    return "[" + productId + " ] " + productName + " Qty: " + quantity + ", Price: ₹" +
price;
}

```

```

}

public class InventoryManager {

    private HashMap<String, Product> inventory = new HashMap<>();

    // Add a product

    public void addProduct(Product product) {

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

    }

    // Update product quantity and price

    public void updateProduct(String productId, int newQuantity, double newPrice) {

        Product product = inventory.get(productId);

        if (product != null) {

            product.setQuantity(newQuantity);

            product.setPrice(newPrice);

        } else {

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

        }

    }

    // Delete a product

    public void deleteProduct(String productId) {

        if (inventory.containsKey(productId)) {

            inventory.remove(productId);

        } else {

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

        }

    }

    // Display all products

    public void displayInventory() {

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

```

```

        System.out.println(p);
    }
}

}

public class Main {

    public static void main(String[] args) {

        // Create InventoryManager instance
        InventoryManager manager = new InventoryManager();

        // Create some products
        Product p1 = new Product("P001", "Laptop", 10, 70000);
        Product p2 = new Product("P002", "Wireless Mouse",30, 1650);
        Product p3 = new Product("P003", "Keyboard", 20, 3375);

        // Add products to inventory
        manager.addProduct(p1);
        manager.addProduct(p2);
        manager.addProduct(p3);

        // Display current inventory
        System.out.println("\n--- Initial Inventory ---");
        manager.displayInventory();

        // Update a product
        manager.updateProduct("P001", 8, 59999.99);

        // Delete a product
        manager.deleteProduct("P002");

        // Display updated inventory

```

```

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

        manager.displayInventory();

    }

}

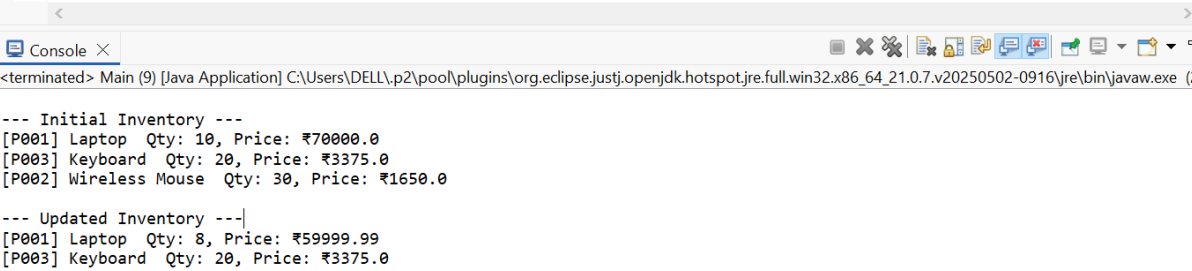
```

## Output:

```

26      // Display updated inventory
27      System.out.println("\n--- Updated Inventory ---");
28      manager.displayInventory();
29  }

```



```

<terminated> Main (9) [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.7.v20250502-0916\jre\bin\javaw.exe (2

--- Initial Inventory ---
[P001] Laptop Qty: 10, Price: ₹70000.0
[P003] Keyboard Qty: 20, Price: ₹3375.0
[P002] Wireless Mouse Qty: 30, Price: ₹1650.0

--- Updated Inventory ---
[P001] Laptop Qty: 8, Price: ₹59999.99
[P003] Keyboard Qty: 20, Price: ₹3375.0

```

## Exercise 2: E-commerce Platform Search Function

1) Explain Big O notation and how it helps in analyzing algorithms.

**Big O notation** is a way to describe how the **time or space complexity** of an algorithm grows as the **input size increases**. It gives us a **high-level idea** of the algorithm's performance and efficiency without focusing on hardware or exact timings.

**Compares algorithms:** Big O lets you compare different algorithms to decide which one will perform better as the data grows.

**Predicts scalability:** It tells you how an algorithm will behave with larger inputs (e.g., 100 items vs. 1 million).

**Highlights inefficiencies:** Helps identify if an algorithm is too slow or memory-heavy for large datasets.

**Ignores machine differences:** Instead of worrying about processor speed or RAM, Big O focuses on the *core logic* of the algorithm.

2) Describe the best, average, and worst-case scenarios for search operations.

**Linear Search** (unsorted list):

- **Best:** Item is first → **O(1)**

- **Average:** Item is in the middle →  **$O(n)$**
  - **Worst:** Item is last or missing →  **$O(n)$**
- Binary Search** (sorted list):
- **Best:** Item is in the middle →  **$O(1)$**
  - **Average:** Found after a few cuts →  **$O(\log n)$**
  - **Worst:** Not found →  **$O(\log n)$**

### Code:

```
package ecommerce;

public class Product {
    int productId;
    String productName;
    String category;

    public Product(int productId, String productName, String category) {
        this.productId = productId;
        this.productName = productName;
        this.category = category;
    }

    public String toString() {
        return "[" + productId + "]" + productName + " - " + category;
    }
}

public class SearchUtil {

    public static Product linearSearch(Product[] products, String name) {
        for (Product p : products) {
            if (p.productName.equalsIgnoreCase(name)) {
                return p;
            }
        }
    }
}
```

```

        return null;
    }

    public static Product binarySearch(Product[] products, String name) {
        int left = 0, right = products.length - 1;

        while (left <= right) {
            int mid = (left + right) / 2;

            int compare = products[mid].productName.compareToIgnoreCase(name);

            if (compare == 0)
                return products[mid];

            else if (compare < 0)
                left = mid + 1;

            else
                right = mid - 1;
        }

        return null;
    }

    public static void sortByName(Product[] products) {
        java.util.Arrays.sort(products, (a, b) ->
a.productName.compareToIgnoreCase(b.productName));
    }
}

public class Main {

    public static void main(String[] args) {

        Product[] products = {

            new Product(101, "Laptop", "Electronics"),

            new Product(102, "Shoes", "Fashion"),

            new Product(103, "Coffee Mug", "Home"),

            new Product(104, "Keyboard", "Electronics"),

            new Product(105, "T-Shirt", "Fashion")

        };
    }
}

```

```

        System.out.println(" Linear Search for 'Keyboard:");

        Product result1 = SearchUtil.linearSearch(products, "Keyboard");

        System.out.println(result1 != null ? result1 : "Product not found");

        SearchUtil.sortByName(products);

        System.out.println("\n Binary Search for 'Keyboard:");

        Product result2 = SearchUtil.binarySearch(products, "Keyboard");

        System.out.println(result2 != null ? result2 : "Product not found");

    }

}

```

## OUTPUT:

```

18         System.out.println(result2 != null ? result2 : "Product not found");
19     }
20 }

```

Console X

<terminated> Main (2) [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86\_64\_21.0.7.v20250502-0916\jre\bin\javaw.exe (21

Linear Search for 'Keyboard':  
[104] Keyboard - Electronics

Binary Search for 'Keyboard':  
[104] Keyboard - Electronics

## Ananlysis:

Aspect	Linear Search	Binary Search
Time Complexity	$O(n)$	$O(\log n)$
Search Speed	Slower for large data	Very fast for large, sorted data
Best Use Case	Small or unsorted product lists	Large product catalogs with sorted entries



### Exercise 3: Sorting Customer Orders

1) Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).

➤ **Bubble Sort**

- Compares two side-by-side elements and swaps them if they're in the wrong order.
- Repeats this process until everything is sorted.
- **Slow** for large lists.

➤ **Insertion Sort**

- Takes one element at a time and puts it in the correct place in the sorted part of the list.
- Works well for small or nearly-sorted lists.
- **Still slow** for large, random lists.

➤ **Quick Sort**

- Picks a pivot value, puts smaller numbers on the left and bigger ones on the right.
- Then sorts each part the same way.
- **Fast** for most cases.

➤ **Merge Sort**

- Breaks the list into halves, sorts each half, and then merges them together.
- Very reliable and always takes about the same time.
- Uses more memory.

**Code:**

```
public 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;
    }
    public int getOrderId() {
        return orderId;
    }
    public String getCustomerName() {
        return customerName;
    }
}
```

```

    }
    public double getTotalPrice() {
        return totalPrice;
    }
    @Override
    public String toString() {
        return "[" + orderId + " " + customerName + " - ₹" + totalPrice;
    }
}

public class BubbleSort {
    public static void sortOrdersByPrice(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 class QuickSort {
    public static void sortOrdersByPrice(Order[] orders, int low, int high) {
        if (low < high) {
            int pi = partition(orders, low, high);
            sortOrdersByPrice(orders, low, pi - 1);
            sortOrdersByPrice(orders, pi + 1, high);
        }
    }

    private static int partition(Order[] orders, int low, int high) {
        double pivot = orders[high].getTotalPrice(); // pivot value
        int i = (low - 1); // index of smaller element
        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;
    }
}

public class Main {
    public static void main(String[] args) {
        // Sample array of orders
        Order[] orders = {
            new Order(101, "Alice", 2500.0),
            new Order(102, "Bob", 1800.0),
            new Order(103, "Charlie", 3200.0),
            new Order(104, "Daisy", 1500.0)
        };
        System.out.println("Original Orders:");
        printOrders(orders);

        // Bubble Sort
        System.out.println("\nSorted using Bubble Sort:");
        BubbleSort.sortOrdersByPrice(orders);
        printOrders(orders);

        // Reset original array for Quick Sort
        Order[] orders2 = {
            new Order(101, "Alice", 2500.0),
            new Order(102, "Bob", 1800.0),
            new Order(103, "Charlie", 3200.0),
            new Order(104, "Daisy", 1500.0)
        };
        System.out.println("\nSorted using Quick Sort:");
        QuickSort.sortOrdersByPrice(orders2, 0, orders2.length - 1);
        printOrders(orders2);
    }

    // Helper method to print orders
    public static void printOrders(Order[] orders) {
        for (Order order : orders) {
            System.out.println(order);
        }
    }
}

```

## OUTPUT:

```
22         new Order(103, "Charlie", 3200.0),
23         new Order(104, "Daisy", 1500.0)
24     };
25     System.out.println("\nSorted using Quick Sort:");
26     QuickSort.sortOrdersByPrice(orders2, 0, orders2.length - 1);
27     printOrders(orders2);
28 }
```

<terminated> Main (10) [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.fu

Original Orders:  
[101] Alice - ₹2500.0  
[102] Bob - ₹1800.0  
[103] Charlie - ₹3200.0  
[104] Daisy - ₹1500.0

Sorted using Bubble Sort:  
[104] Daisy - ₹1500.0  
[102] Bob - ₹1800.0  
[101] Alice - ₹2500.0  
[103] Charlie - ₹3200.0

Sorted using Quick Sort:  
[104] Daisy - ₹1500.0  
[102] Bob - ₹1800.0  
[101] Alice - ₹2500.0  
[103] Charlie - ₹3200.0

## Analysis:

Algorithm	Best Case	Average Case	Worst Case	Efficiency
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Very slow for large data
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$ (rare)	Very fast and efficient

## Why Quick Sort is Preferred Over Bubble Sort:

- Much faster on average, especially for large datasets
- Efficient divide-and-conquer strategy
- Less swapping compared to Bubble Sort

## Exercise 4: Employee Management System

### Memory Representation of Arrays:

- Arrays are contiguous blocks of memory where each element is of the same data type.
- The base address of the array points to the first element.
- Indexing uses an offset:  
$$\text{address} = \text{base\_address} + (\text{index} \times \text{size\_of\_element})$$

### Advantages:

- Fast access using index:  $O(1)$  time
- Efficient for fixed-size data.
- Memory locality enhances performance in iteration.

### Code:

```
import java.util.Scanner;

public 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;
    }

    public int getEmployeeId() {
        return employeeId;
    }

    public String toString() {
```

```

        return "[" + employeeId + "] " + name + " - " + position + " - ₹" + salary;
    }
}

public class EmployeeManager {

    private Employee[] employees;
    private int count;

    public EmployeeManager(int size) {
        employees = new Employee[size];
        count = 0;
    }

    // Method to add a new employee
    public void addEmployee(Employee emp) {
        if (count < employees.length) {
            employees[count++] = emp;
            System.out.println("Employee added successfully.");
        } else {
            System.out.println("Employee list is full.");
        }
    }

    // Method to display all employees
    public void displayEmployees() {
        if (count == 0) {
            System.out.println("No employees to display.");
            return;
        }

        System.out.println("Employee List:");
        for (int i = 0; i < count; i++) {
            System.out.println(employees[i]);
        }
    }
}

```

```

    }
}

// Method to search employee by ID
public void searchEmployee(int empId) {
    for (int i = 0; i < count; i++) {
        if (employees[i].getEmployeeId() == empId) {
            System.out.println("Employee Found: " + employees[i]);
            return;
        }
    }

    System.out.println("Employee with ID " + empId + " not found.");
}

// Method to delete employee by ID
public void deleteEmployee(int empId) {
    for (int i = 0; i < count; i++) {
        if (employees[i].getEmployeeId() == empId) {
            // Shift elements to the left
            for (int j = i; j < count - 1; j++) {
                employees[j] = employees[j + 1];
            }
            employees[--count] = null;
            System.out.println("Employee deleted successfully.");
            return;
        }
    }

    System.out.println("Employee with ID " + empId + " not found.");
}
}

```

```
public class Main {  
    public static void main(String[] args) {  
        EmployeeManager manager = new EmployeeManager(10); // Max 10 employees  
        Scanner sc = new Scanner(System.in);  
        int choice;  
        do {  
            System.out.println("\n--- Employee Management System ---");  
            System.out.println("1. Add Employee");  
            System.out.println("2. Display Employees");  
            System.out.println("3. Search Employee");  
            System.out.println("4. Delete Employee");  
            System.out.println("5. Exit");  
            System.out.print("Enter your choice: ");  
            choice = sc.nextInt();  
            sc.nextLine();  
            switch (choice) {  
                case 1:  
                    System.out.print("Enter ID: ");  
                    int id = sc.nextInt();  
                    sc.nextLine();  
                    System.out.print("Enter Name: ");  
                    String name = sc.nextLine();  
                    System.out.print("Enter Position: ");  
                    String position = sc.nextLine();  
                    System.out.print("Enter Salary: ");  
                    double salary = sc.nextDouble();  
                    manager.addEmployee(new Employee(id, name, position, salary));  
                    break;
```



```
case 2:
    manager.displayEmployees();
    break;
case 3:
    System.out.print("Enter ID to search: ");
    int searchId = sc.nextInt();
    manager.searchEmployee(searchId);
    break;
case 4:
    System.out.print("Enter ID to delete: ");
    int deleteId = sc.nextInt();
    manager.deleteEmployee(deleteId);
    break;
case 5:
    System.out.println("Exiting system. Goodbye!");
    break;
default:
    System.out.println("Invalid choice. Try again.");
}
} while (choice != 5);
sc.close();
}
}
```

## OUTPUT:

```
5 Scanner sc = new Scanner(System.in);
6 int choice;
7 do {
8     System.out.println("\n--- Employee Management System ---");
9     System.out.println("1. Add Employee");
10    System.out.println("2. Display Employees");

```

Console X

Main (11) [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86\_64\_21.0.7.v20250502-0916\jre\bin\java

```
--- Employee Management System ---
1. Add Employee
2. Display Employees
3. Search Employee
4. Delete Employee
5. Exit
Enter your choice: 1
Enter ID: 101
Enter Name: bob
Enter Position: 2
Enter Salary: 20000
Employee added successfully.

--- Employee Management System ---
1. Add Employee
2. Display Employees
3. Search Employee
4. Delete Employee
5. Exit
Enter your choice:
```

## Analysis:

Operation	Time Complexity	Explanation
Add	$O(1)$	Employee is added at the end using the count index — no shifting needed.
Search	$O(n)$	Linear search is used to find the employee by ID.
Traverse	$O(n)$	All employee records are printed one by one.
Delete	$O(n)$	After finding the employee, elements are shifted to fill the gap.

## Limitations of Arrays:

- 1.Fixed Size
- 2.Slow Deletion and Insertion (in middle)
- 3.Linear Search
- 4.No Built-in Dynamic Features

## Exercise 5: Task Management System

1) Explain the different types of linked lists (Singly Linked List, Doubly Linked List).

### 1. Singly Linked List:

- Each node has two parts: data and next.
- The next points to the next node in the list.
- Traversal is possible in only one direction.

### 2. Doubly Linked List:

- Each node has three parts: prev, data, and next.
- Allows traversal in both forward and backward directions.
- More memory is required due to the extra prev pointer.

## OUTPUT:

```
13 Task t = manager.searchTask(2);

<terminated> Main (12) [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.7.v2025050C
Task added: [1] Design UI - Pending
Task added: [2] Write Backend - In Progress
Task added: [3] Test System - Pending

All Tasks:
[1] Design UI - Pending
[2] Write Backend - In Progress
[3] Test System - Pending

Searching Task with ID 2:
[2] Write Backend - In Progress

Deleting Task with ID 1:
Task deleted: [1] Design UI - Pending

All Tasks After Deletion:
[2] Write Backend - In Progress
[3] Test System - Pending
```

## Analysis:

Operation	Time Complexity	Explanation
Add Task	$O(n)$	Traverses to end to insert
Search Task	$O(n)$	Needs to traverse each node
Delete Task	$O(n)$	Must find the node and adjust pointers
Display Tasks	$O(n)$	Visits each node once

## Advantages of Linked Lists over Arrays:

- Dynamic size: No need to define fixed size.
- Efficient Insert/Delete:  $O(1)$  insertion/deletion at head.
- Memory efficient: No need to allocate extra memory (unlike arrays which may waste space).

## Exercise 6: Library Management System

1) Explain linear search and binary search algorithms.

### Linear Search:

- Definition: Scans each element in the list one by one.
- Use Case: Works on *unsorted* or *small* datasets.
- Time Complexity:
  - Best:  $O(1)$
  - Average/Worst:  $O(n)$

### Binary Search:

- Definition: Divides the sorted list in half each time to search.
- Use Case: Only works on *sorted* datasets.
- Time Complexity:
  - Best:  $O(1)$
  - Average/Worst:  $O(\log n)$

### Code:

```
import java.util.List;

import java.util.ArrayList;

import java.util.Scanner;

public class Book {

    private int bookId;

    private String title;
```

```

private String author;

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

    this.bookId = bookId;

    this.title = title;

    this.author = author;
}

public int getBookId() {

    return bookId;
}

public String getTitle() {

    return title;
}

public String getAuthor() {

    return author;
}

@Override

public String toString() {

    return "[" + bookId + "]" + title + " by " + author;
}
}

public class SearchUtil {

    public static Book linearSearchByTitle(List<Book> books, String title) {

        for (Book book : books) {

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

                return book;

            }

        }

        return null;
    }
}

```

```

}

public class Library {

    public static void main(String[] args) {

        List<Book> books = new ArrayList<>();

        books.add(new Book(1, "The Alchemist", "Paulo Coelho"));

        books.add(new Book(2, "1984", "George Orwell"));

        books.add(new Book(3, "To Kill a Mockingbird", "Harper Lee"));

        books.add(new Book(4, "Brave New World", "Aldous Huxley"));

        books.add(new Book(5, "The Great Gatsby", "F. Scott Fitzgerald"));

        Scanner scanner = new Scanner(System.in);

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

        String searchTitle = scanner.nextLine();

        Book foundBook = SearchUtil.linearSearchByTitle(books, searchTitle);

        if (foundBook != null) {

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

        } else {

            System.out.println("Book not found.");

        }

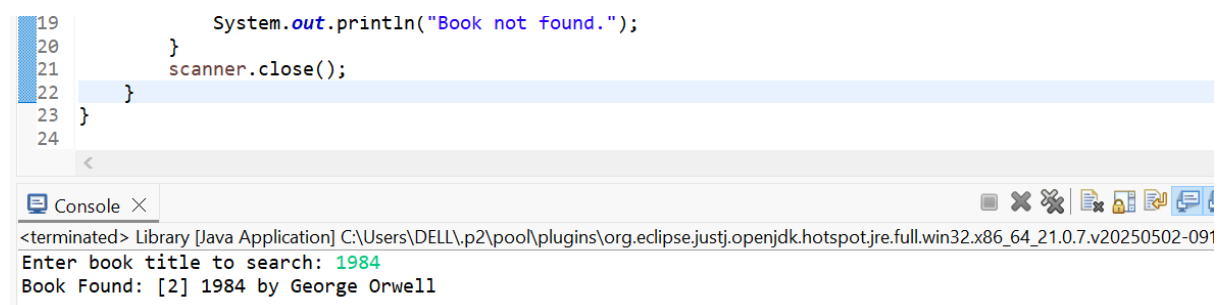
        scanner.close();

    }

}

```

## OUTPUT:



The screenshot shows an IDE with a code editor and a console window. The code editor displays lines 19 to 24 of the program, with line 20 highlighted. The console window shows the output of the program, including the prompt 'Enter book title to search: 1984' and the result 'Book Found: [2] 1984 by George Orwell'.

```

19         System.out.println("Book not found.");
20     }
21     scanner.close();
22 }
23 }
24

```

Console Output:

```

<terminated> Library [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.7.v20250502-091
Enter book title to search: 1984
Book Found: [2] 1984 by George Orwell

```

## Exercise 7: Financial Forecasting

1) Explain the concept of recursion and how it can simplify certain problems

Recursion is a technique where a function calls itself to solve a smaller version of a problem.

**It has two parts:**

1. Base case – The stopping condition (when to stop calling itself).
2. Recursive case – The part where the function calls itself.

**How Recursion Simplifies Problems:**

**Recursion is useful when:**

- The problem can be broken into smaller repeating steps.
- Each step is similar to the previous one.
- **Examples:**
  - Calculating factorials (n!)
  - Fibonacci numbers
  - Searching in trees
  - File system traversal
  - Financial forecasting (predicting future values year by year)

**Code:**

```
package finance;

public class Main {

    public static void main(String[] args) {

        double initialInvestment = 10000.0;

        double annualGrowthRate = 0.08;

        int forecastYears = 5;


        double resultRecursive = Forecast.futureValueRecursive(initialInvestment,
annualGrowthRate, forecastYears);

        System.out.printf("Recursive Forecast after %d years: ₹%.2f\n", forecastYears,
resultRecursive);
```

```

        double[] memo = new double[forecastYears + 1];

        double resultMemo = Forecast.futureValueMemo(initialInvestment, annualGrowthRate,
forecastYears, memo);

        System.out.printf("Optimized Forecast after %d years: ₹%.2f\n", forecastYears,
resultMemo);
    }
}

public class Main {

    public static void main(String[] args) {

        double initialInvestment = 10000.0;

        double annualGrowthRate = 0.08;

        int forecastYears = 5;

        double resultRecursive = Forecast.futureValueRecursive(initialInvestment,
annualGrowthRate, forecastYears);

        System.out.printf("Recursive Forecast after %d years: ₹%.2f\n", forecastYears,
resultRecursive);

        double[] memo = new double[forecastYears + 1];

        double resultMemo = Forecast.futureValueMemo(initialInvestment, annualGrowthRate,
forecastYears, memo);

        System.out.printf("Optimized Forecast after %d years: ₹%.2f\n", forecastYears,
resultMemo);
    }
}

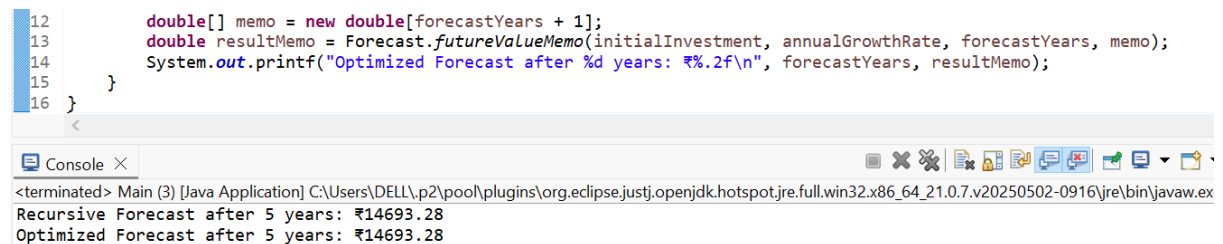
```

## OUTPUT:

```

12         double[] memo = new double[forecastYears + 1];
13         double resultMemo = Forecast.futureValueMemo(initialInvestment, annualGrowthRate, forecastYears, memo);
14         System.out.printf("Optimized Forecast after %d years: ₹%.2f\n", forecastYears, resultMemo);
15     }
16 }

```



Console

<terminated> Main (3) [Java Application] C:\Users\DELL\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86\_64\_21.0.7.v20250502-0916\jre\bin\javaw.exe

Recursive Forecast after 5 years: ₹14693.28

Optimized Forecast after 5 years: ₹14693.28



## **Analysis:**

### **futureValueRecursive (Plain Recursion)**

- Time Complexity:  $O(n)$
- Space Complexity:  $O(n)$  (due to call stack)
- Drawback: Risk of stack overflow for large years.

### **futureValueMemo (With Memoization)**

- Time Complexity:  $O(n)$
- Space Complexity:  $O(n)$  (uses memo array + call stack)
- Advantage: Avoids repeated calculations, faster than plain recursion.