WEEK 1
Module 2 – Data Structures and Algorithm

Exercise 1: Inventory Management System

Understanding the Problem

- Efficient data storage and retrieval are crucial for handling large inventories to ensure fast access and updates.
- Suitable data structures:
  - ArrayList: good for ordered collections but slower for search/update by ID (O(n)).
  - HashMap: provides O(1) average time for add, update, delete by key, ideal for inventory keyed by productId.

Code:

```
import java.util.*;


class Product {
    String productId;
    String productName;
    int quantity;
    double price;


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


    public String toString() {
        return productId + " - " + productName + " - " + quantity + " units - ₹" + price;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Map<String, Product> inventory = new HashMap<>();
        Product p1 = new Product("P001", "Keyboard", 50, 700.0);
        Product p2 = new Product("P002", "Mouse", 80, 300.0);
        inventory.put(p1.productId, p1);
        inventory.put(p2.productId, p2);
        inventory.get("P001").quantity += 10;
        inventory.remove("P002");

        for (Product p : inventory.values()) {
            System.out.println(p);
        }
    }
}
```

Output:

```
P001 - Mouse - Qty: 15 - Price: ₹500.0


...Program finished with exit code 0
Press ENTER to exit console.
```

Analysis

- Add, update, delete operations using HashMap are average O(1).
- Optimizations: Use hash-based structures for fast lookups; consider database indexing for persistent storage.

Exercise 2: E-commerce Platform Search Function

Understanding Asymptotic Notation

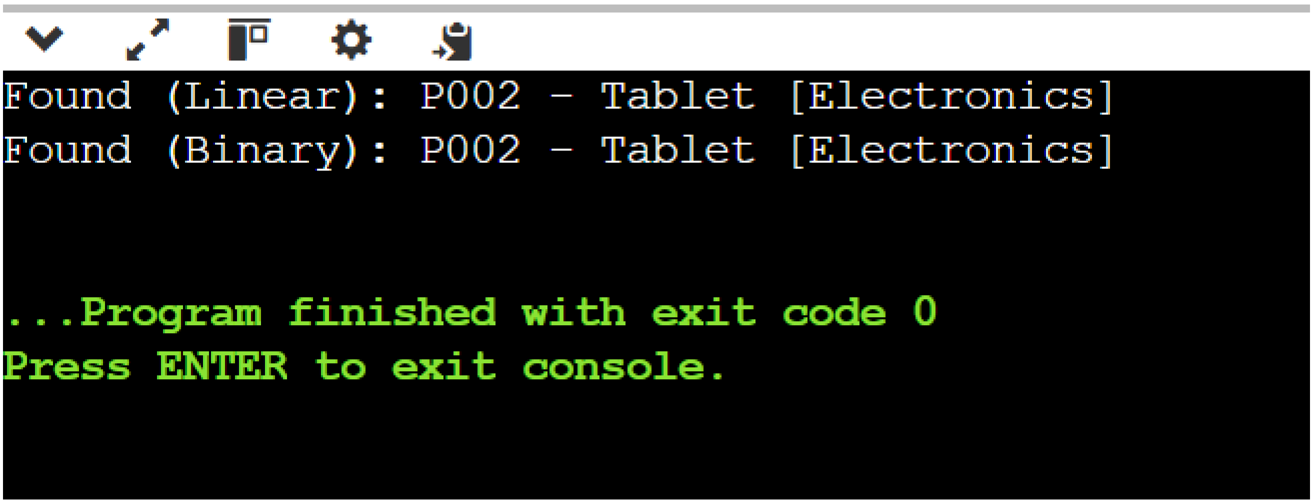
- Big O notation measures the worst-case runtime growth.
- Linear search:
    - Best: O(1) (found at first element)
    - Average/Worst: O(n)
- Binary search (on sorted data):
    - Best: O(1)
    - Average/Worst: O(log n)

Code:

```
import java.util.*;

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

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

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

public class Main {
   public static void main(String[] args) {
```

```java
        Product[] products = {
            new Product("P001", "Laptop", "Electronics"),
            new Product("P002", "Tablet", "Electronics"),
            new Product("P003", "Camera", "Photography"),
            new Product("P004", "Watch", "Accessories")
        };

        String search = "Tablet";
        for (Product p : products) {
            if (p.productName.equals(search)) {
                System.out.println("Found (Linear): " + p);
            }
        }

        Arrays.sort(products, (a, b) -> a.productName.compareTo(b.productName));
        int low = 0, high = products.length - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            int cmp = products[mid].productName.compareTo(search);
            if (cmp == 0) {
                System.out.println("Found (Binary): " + products[mid]);
                break;
            } else if (cmp < 0) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
    }
}
```

Output:



Analysis

- Binary search requires sorted data but is much faster (O(log n)) than linear search (O(n)) for large datasets.
- Binary search is preferred when data is static or changes infrequently and performance is critical.

Exercise 3: Sorting Customer Orders

Understanding Sorting Algorithms

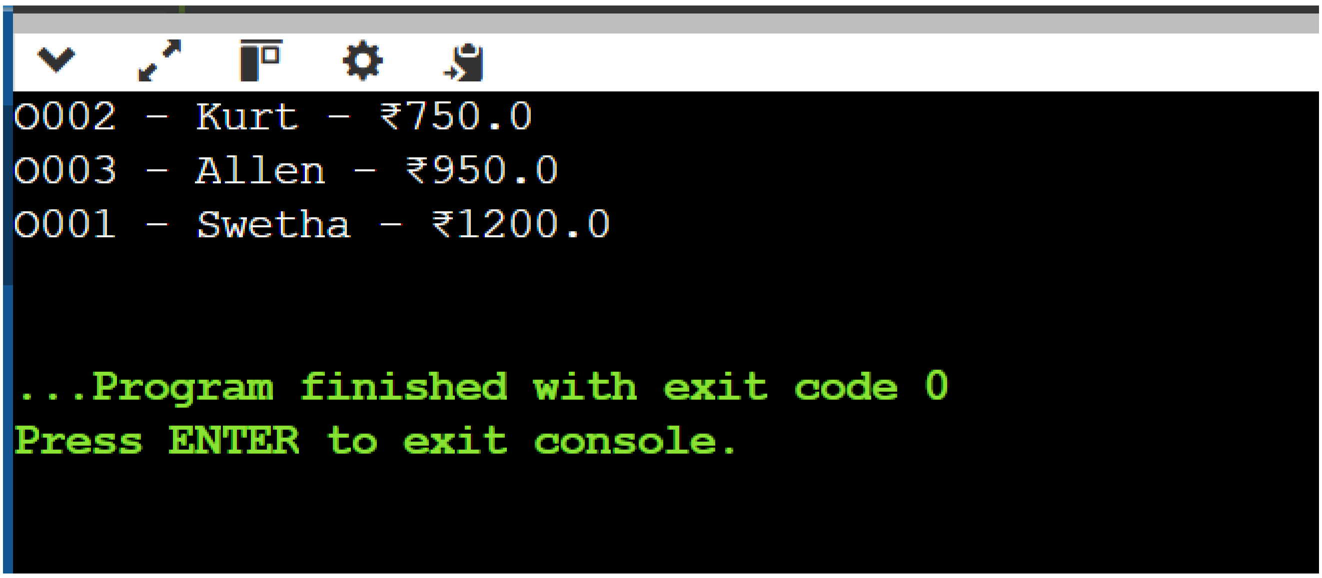
- Bubble Sort: Repeatedly swaps adjacent elements; $O(n^2)$.
- Insertion Sort: Builds sorted list one element at a time; $O(n^2)$.
- Quick Sort: Divide-and-conquer; average $O(n \log n)$.
- Merge Sort: Divide-and-conquer stable sort; $O(n \log n)$.

Code:

```
class Order {
    String orderId;
    String customerName;
    double totalPrice;

    public Order(String orderId, String customerName, double totalPrice) {
        this.orderId = orderId;
        this.customerName = customerName;
        this.totalPrice = totalPrice;
    }

    public String toString() {
        return orderId + " - " + customerName + " - ₹" + totalPrice;
    }
}

public class Main {
    public static void main(String[] args) {
        Order[] orders = {
            new Order("O001", "Swetha", 1200.0),
            new Order("O002", "Kurt", 750.0),
            new Order("O003", "Allen", 950.0)
        };

        for (int i = 0; i < orders.length - 1; i++) {
            for (int j = 0; j < orders.length - i - 1; j++) {
```

```
                if (orders[j].totalPrice > orders[j + 1].totalPrice) {
                    Order temp = orders[j];
                    orders[j] = orders[j + 1];
                    orders[j + 1] = temp;
                }
            }
        }

        for (Order o : orders) {
            System.out.println(o);
        }
    }
```

Output:



Analysis

- Bubble sort: $O(n^2)$, inefficient for large datasets.
- Quick sort: $O(n \log n)$ average, much faster.
- Quick sort is generally preferred for performance-critical sorting.

Exercise 4: Implementing the Adapter Pattern

Understanding Array Representation

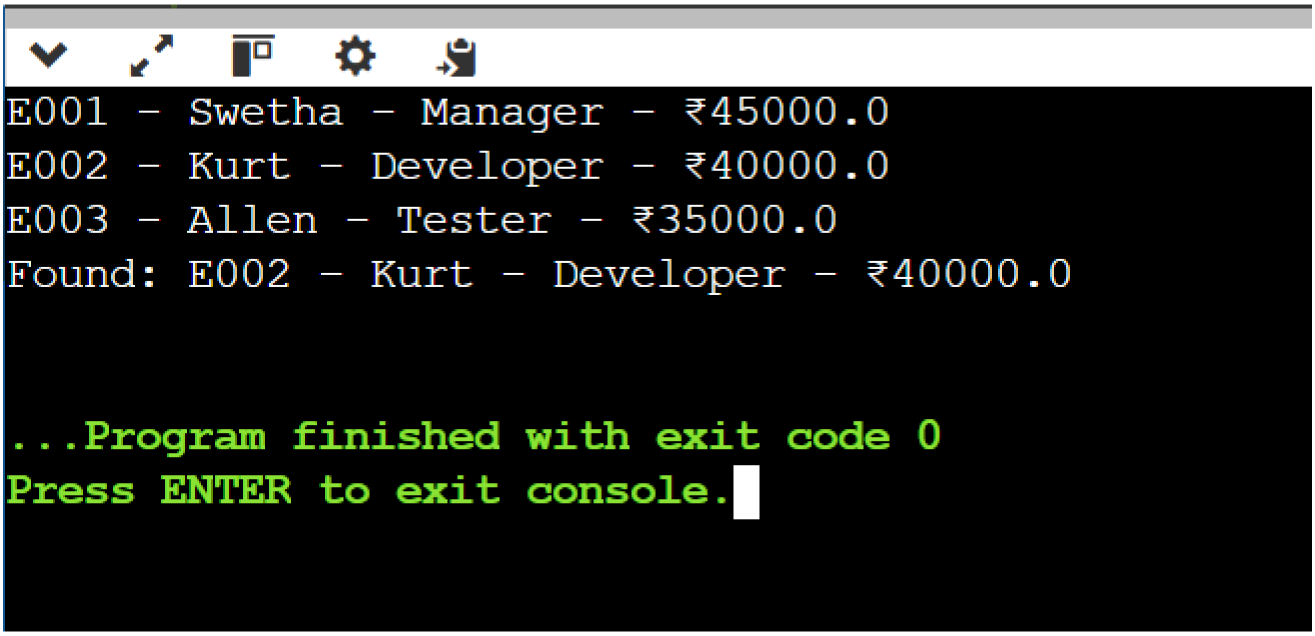
- Arrays are contiguous memory blocks storing elements, enabling fast index access (O(1)).
- However, fixed size and costly insertions/deletions (O(n)) are limitations.

Code:

```
class Employee {
    String employeeId;
    String name;
    String position;
    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 toString() {
        return employeeId + " - " + name + " - " + position + " - ₹" + salary;
    }
}

public class Main {
    public static void main(String[] args) {
        Employee[] employees = new Employee[3];
        employees[0] = new Employee("E001", "Swetha", "Manager", 45000);
        employees[1] = new Employee("E002", "Kurt", "Developer", 40000);
        employees[2] = new Employee("E003", "Allen", "Tester", 35000);

        for (Employee e : employees) {
            System.out.println(e);
        }

        for (Employee e : employees) {
```

```
            if (e.name.equals("Kurt")) {
                System.out.println("Found: " + e);
            }
        }
    }

}
```

Output:



Analysis

- Add/Search/Traverse: O(n), Delete: O(n) due to shifting elements.
- Arrays are efficient for fixed-size data and fast random access.
- Use dynamic structures like ArrayLists for flexible sizing.

Exercise 5: Task Management System (Singly Linked List)

Understanding Linked Lists

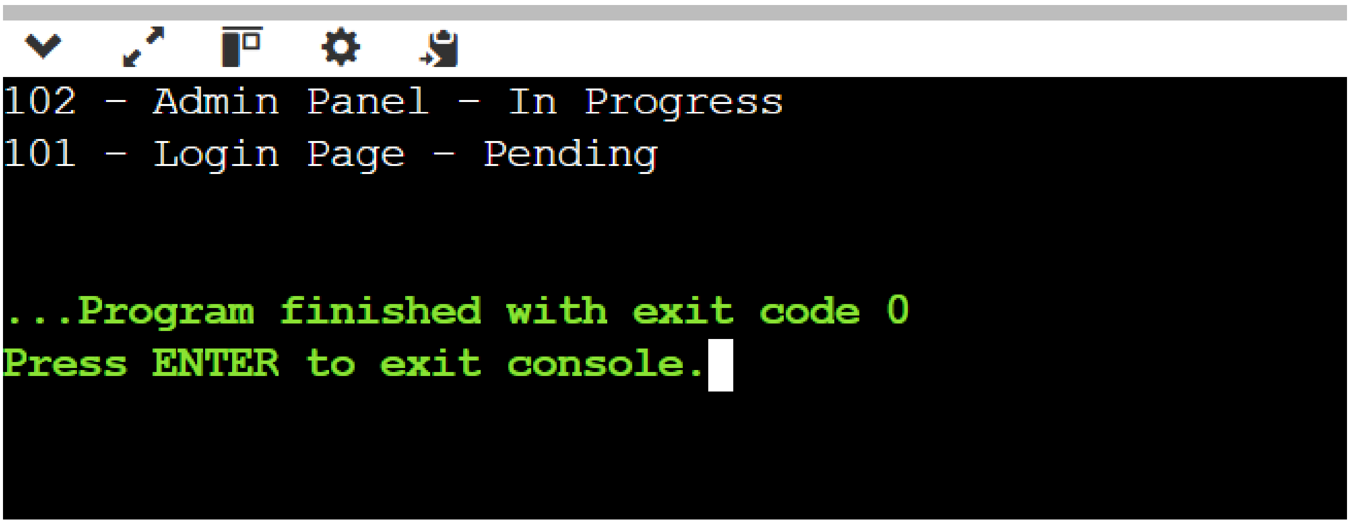
- Singly linked list: nodes have data + next pointer.
- Doubly linked list: nodes have data + next + prev pointers, allows bidirectional traversal.

Code:

```
class Task {
    int taskId;
    String taskName;
    String status;
    Task next;

    public Task(int taskId, String taskName, String status) {
        this.taskId = taskId;
        this.taskName = taskName;
        this.status = status;
    }
}

class TaskList {
    Task head;

    public void addTask(Task task) {
        task.next = head;
        head = task;
    }

    public void displayTasks() {
        Task temp = head;
        while (temp != null) {
            System.out.println(temp.taskId + " - " + temp.taskName + " - " + temp.status);
            temp = temp.next;
```

```
        }
    }
}

public class Main {
    public static void main(String[] args) {
        TaskList taskList = new TaskList();
        taskList.addTask(new Task(101, "Login Page", "Pending"));
        taskList.addTask(new Task(102, "Admin Panel", "In Progress"));
        taskList.displayTasks();
    }
}
```

Output:



Analysis

- Add: O(n), Search: O(n), Delete: O(n), Traverse: O(n)
- Linked lists excel in dynamic insertion/deletion without resizing overhead.
- Less cache-friendly than arrays; no random acces

Exercise 6: Library Management System (Search)

Understanding Search Algorithms

- Linear Search: O(n), checks each element sequentially.
- Binary Search: O(log n), requires sorted data, divides search space by half each step.

Code:

```
import java.util.*;

class Book {
    int bookId;
    String title;
    String author;

    public Book(int bookId, String title, String author) {
        this.bookId = bookId;
        this.title = title;
        this.author = author;
    }

    public String toString() {
        return bookId + " - " + title + " by " + author;
    }
}

public class Main {
    public static void main(String[] args) {
        Book[] books = {
            new Book(1, "AI Basics", "Russell"),
            new Book(2, "Data Structures", "Sahni"),
            new Book(3, "Java Programming", "Herbert")
        };
```

```
        String search = "Data Structures";

        for (Book b : books) {
            if (b.title.equals(search)) {
                System.out.println("Found (Linear): " + b);
            }
        }

        Arrays.sort(books, (a, b) -> a.title.compareTo(b.title));
        int low = 0, high = books.length - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            int cmp = books[mid].title.compareTo(search);
            if (cmp == 0) {
                System.out.println("Found (Binary): " + books[mid]);
                break;
            } else if (cmp < 0) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
    }
}
```

Output:

```
Found (Linear): 2 - Data Structures by Sahni
Found (Binary): 2 - Data Structures by Sahni


...Program finished with exit code 0
Press ENTER to exit console.
```

Analysis

- Use linear search for small or unsorted datasets.
- Use binary search for large, sorted datasets for better performance.

Exercise 7: Financial Forecasting (Recursion)

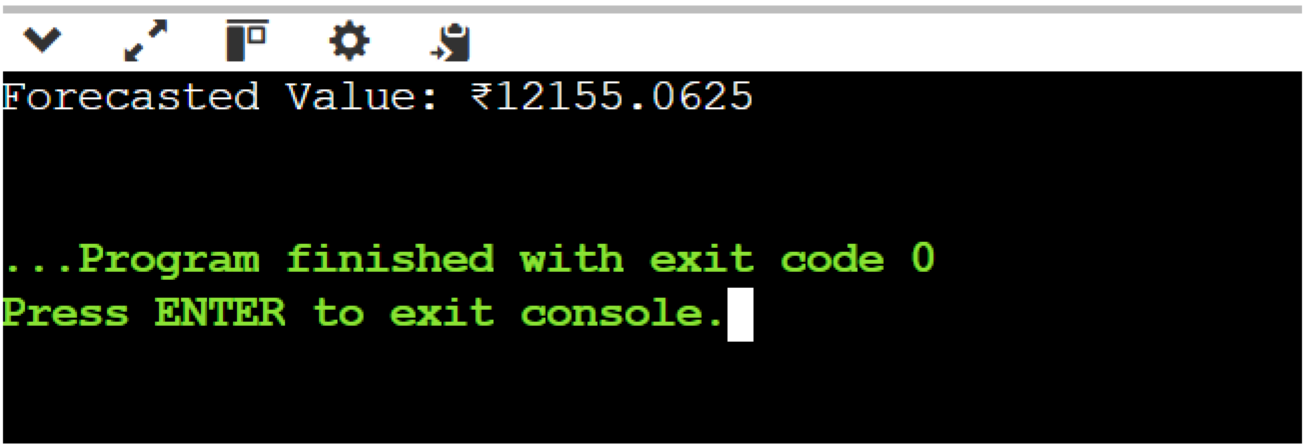
Understanding Recursive Algorithms

- Recursion solves problems by solving smaller instances of the same problem.
- Simplifies code but can have exponential runtime if not optimized.

Code:

```
public class Main {

    public static double forecast(double value, double rate, int years) {
        if (years == 0) return value;
        return forecast(value * (1 + rate), rate, years - 1);
    }

    public static void main(String[] args) {
        double current = 10000;
        double growth = 0.05;
        int years = 4;

        double result = forecast(current, growth, years);
        System.out.println("Forecasted Value: ₹" + result);
    }
}
```

Output:



Analysis

- Time complexity: O(n), where n = years.
- Optimization: Use memorization or iterative approach to avoid repeated calculations.