Algorithms and Data Structures:

Exercise 1:

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

Ans. Data structures and algorithms are essential in handling large inventories because they enable efficient data management, quick retrieval, and optimization. They support scalability, accuracy, and automation of inventory processes. Efficient algorithms help in predicting demand and optimizing stock levels. Data structures like trees and hash tables facilitate fast lookup and modification. Overall, they improve inventory management effectiveness.

2. Discuss the types of data structures suitable for this problem

Ans. Arrays, HashTable.

InventoryManager.java:

```
package com.inventory;

import java.util.HashMap;

public class InventoryManager {

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

   public void addProduct(Product product) {

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

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

   }

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

      Product product = inventory.get(productId);

      if (product != null) {

         product.setProductName(name);

         product.setQuantity(quantity);

         product.setPrice(price);
```

```
            System.out.println("Product updated: " + product);
        } else {
            System.out.println("Product not found.");
        }
    }
    public void deleteProduct(String productId) {
        Product removed = inventory.remove(productId);
        if (removed != null) {
            System.out.println("Product removed: " + removed);
        } else {
            System.out.println("Product not found.");
        }
    }
    public void displayInventory() {
        System.out.println("\nCurrent Inventory:");
        for (Product p : inventory.values()) {
            System.out.println(p);
        }
    }
}
```

Products.java:

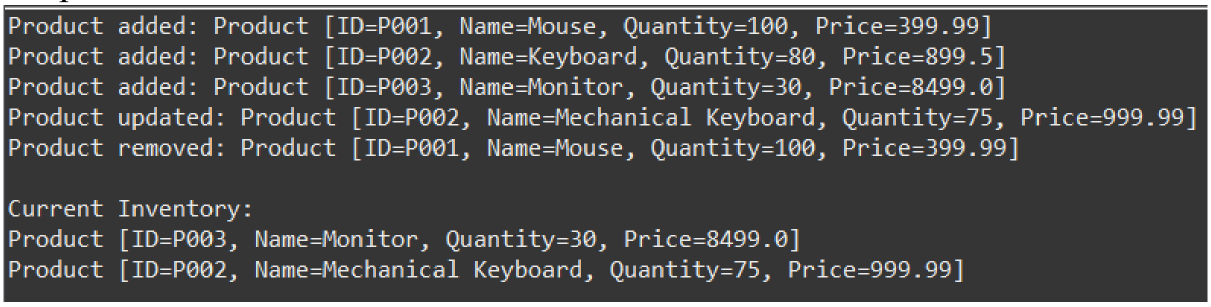
```
package com.inventory;
public class Product {
      private String productId;
         private String productName;
         private int quantity;
         private double price;
```

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

        public String getProductId() { return productId; }
        public String getProductName() { return productName; }
        public int getQuantity() { return quantity; }
        public double getPrice() { return price; }

        public void setProductName(String productName) { this.productName
= productName; }
        public void setQuantity(int quantity) { this.quantity = quantity; }
        public void setPrice(double price) { this.price = price; }

        @Override
        public String toString() {
            return "Product [ID=" + productId + ", Name=" + productName +
                   ", Quantity=" + quantity + ", Price=" + price + "]";
        }
    }
```

Main.java:

```
package com.inventory;

public class Main {
    public static void main(String[] args) {
        InventoryManager manager = new InventoryManager();
        Product p1 = new Product("P001", "Mouse", 100, 399.99);
        Product p2 = new Product("P002", "Keyboard", 80, 899.50);
        Product p3 = new Product("P003", "Monitor", 30, 8499.00);
        manager.addProduct(p1);
        manager.addProduct(p2);
        manager.addProduct(p3);
        manager.updateProduct("P002", "Mechanical Keyboard", 75, 999.99);
        manager.deleteProduct("P001");
        manager.displayInventory();
    }
}
```

Output:



```
Product added: Product [ID=P001, Name=Mouse, Quantity=100, Price=399.99]
Product added: Product [ID=P002, Name=Keyboard, Quantity=80, Price=899.5]
Product added: Product [ID=P003, Name=Monitor, Quantity=30, Price=8499.0]
Product updated: Product [ID=P002, Name=Mechanical Keyboard, Quantity=75, Price=999.99]
Product removed: Product [ID=P001, Name=Mouse, Quantity=100, Price=399.99]

Current Inventory:
Product [ID=P003, Name=Monitor, Quantity=30, Price=8499.0]
Product [ID=P002, Name=Mechanical Keyboard, Quantity=75, Price=999.99]
```

Exercise 2 – Ecommerce Platform System

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

Ans. Big O notation describes the time complexity of an algorithm in terms of input size (n). It helps estimate how the algorithm scales as data grows.

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

Ans. Common Cases for Search:

Search Type	Best Case	Average Case	Worst Case
Linear	O(1) (first item)	O(n/2) ≈ O(n)	O(n)
Binary	O(1) (middle)	O(log n)	O(log n)

Binary search requires sorted data.

Product.java:

```
package com.search;


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


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


    public String getProductId() { return productId; }
    public String getProductName() { return productName; }
    public String getCategory() { return category; }
    @Override
    public String toString() {
        return "Product[ID=" + productId + ", Name=" + productName + ",
Category=" + category + "]";
    }
}
```

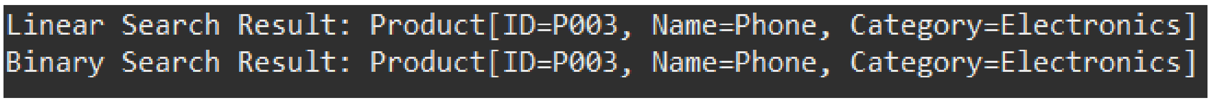
SearchService.java:

```
package com.search;


public class SearchService {


    public static Product linearSearch(Product[] products, String name) {
        for (Product product : products) {
            if (product.getProductName().equalsIgnoreCase(name)) {
                return product;
            }
        }
        return null;
    }


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


        while (left <= right) {
            int mid = (left + right) / 2;
            int comparison =
products[mid].getProductName().compareToIgnoreCase(name);

            if (comparison == 0) {
                return products[mid];
            } else if (comparison < 0) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null;
    }
}
```

Main.java:

```
package com.search;

import java.util.Arrays;
import java.util.Comparator;

public class Main {
    public static void main(String[] args) {
        Product[] products = {
            new Product("P001", "Shoes", "Fashion"),
            new Product("P002", "Watch", "Accessories"),
            new Product("P003", "Phone", "Electronics"),
            new Product("P004", "Laptop", "Electronics")
        };
        Product result1 = SearchService.linearSearch(products, "Phone");
        System.out.println("Linear Search Result: " + result1);

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

        Product result2 = SearchService.binarySearch(products, "Phone");
        System.out.println("Binary Search Result: " + result2);
    }
}
```

Output:



```
Linear Search Result: Product[ID=P003, Name=Phone, Category=Electronics]
Binary Search Result: Product[ID=P003, Name=Phone, Category=Electronics]
```

Exercise 7: Financial Forecasting

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

Ans. Recursion is when a method calls itself to solve a smaller version of the same problem. It is useful when:

- The problem can be broken into subproblems of the same type.
- The subproblems reduce in size and converge to a base case.

Example Use Case: Financial Forecasting

We want to predict future values like this:

Future Value = Current Value × (1 + Growth Rate) applied repeatedly for n periods.

2. Create a method to calculate the future value using a recursive approach.

Ans. currentValue: Starting amount

growthRate: Annual (or periodic) growth rate (e.g., 0.1 for 10%)

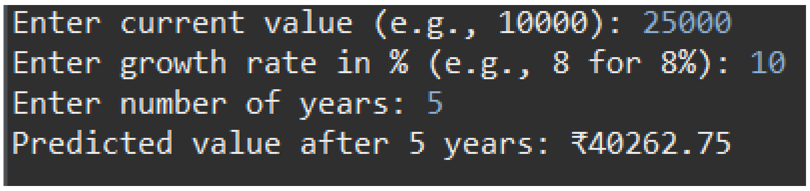
years: Number of years into the future

FinancialForecast.java:

```
package com.financial;

import java.util.Scanner;

public class FinancialForecast {

   public static double predictFutureValue(double currentValue, double
growthRate, int years) {
      if (years == 0) {
         return currentValue;
      }
      return predictFutureValue(currentValue * (1 + growthRate),
growthRate, years - 1);
   }

   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);

      System.out.print("Enter current value (e.g., 10000): ");
      double currentValue = scanner.nextDouble();
```

```
        System.out.print("Enter growth rate in % (e.g., 8 for 8%): ");
        double growthRatePercent = scanner.nextDouble();
        double growthRate = growthRatePercent / 100.0;

        System.out.print("Enter number of years: ");
        int years = scanner.nextInt();

        double futureValue = predictFutureValue(currentValue, growthRate,
years);

        System.out.printf("Predicted value after %d years: ₹%.2f\n", years,
futureValue);

        scanner.close();
    }
}
```

Output:



```
Enter current value (e.g., 10000): 25000
Enter growth rate in % (e.g., 8 for 8%): 10
Enter number of years: 5
Predicted value after 5 years: ₹40262.75
```