**Module 1 - Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

public class SingletonPatternExample {

    static class Logger {

        private static Logger instance;

        private Logger() {

            System.out.println("Logger initialized");

        }

        public static Logger getInstance() {

            if (instance == null) {

                instance = new Logger();

            }

            return instance;

        }

        public void log(String message) {

            System.out.println("[LOG]: " + message);

        }

    }

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

        logger1.log("Logging first message");

        logger2.log("Logging second message");

        if (logger1 == logger2) {

            System.out.println("Both logger1 and logger2 are the same instance.");

        } else {

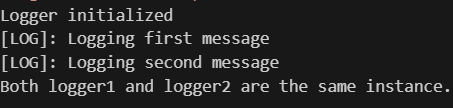
            System.out.println("Different logger instances exist!");

        }

    }

}

OUTPUT:



**Exercise 2: Implementing the Factory Method Pattern**

public class FactoryMethodPatternExample {

    interface Document {

        void open();

    }

    static class WordDocument implements Document {

        public void open() {

            System.out.println("Opening a Word document.");

        }

    }

    static class PdfDocument implements Document {

        public void open() {

            System.out.println("Opening a PDF document.");

        }

    }

    static class ExcelDocument implements Document {

        public void open() {

            System.out.println("Opening an Excel document.");

        }

    }

    static abstract class DocumentFactory {

        public abstract Document createDocument();

    }

    static class WordDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new WordDocument();

        }

    }

    static class PdfDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new PdfDocument();

        }

    }

    static class ExcelDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new ExcelDocument();

        }

    }

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        Document pdfDoc = pdfFactory.createDocument();

        Document excelDoc = excelFactory.createDocument();

        wordDoc.open();

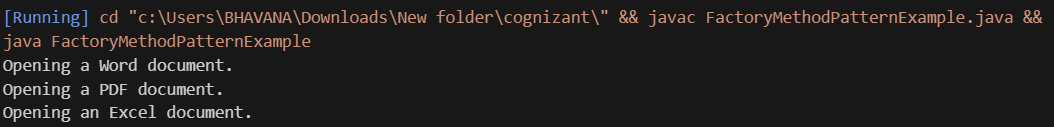
        pdfDoc.open();

        excelDoc.open();

    }

}

OUTPUT:



**Exercise 3: Implementing the Builder Pattern**

public class BuilderPatternExample {

    static class Computer {

        private String CPU;

        private String RAM;

        private String storage;

        private Computer(Builder builder) {

            this.CPU = builder.CPU;

            this.RAM = builder.RAM;

            this.storage = builder.storage;

        }

        public static class Builder {

            private String CPU;

            private String RAM;

            private String storage;

            public Builder setCPU(String CPU) {

                this.CPU = CPU;

                return this;

            }

            public Builder setRAM(String RAM) {

                this.RAM = RAM;

                return this;

            }

            public Builder setStorage(String storage) {

                this.storage = storage;

                return this;

            }

            public Computer build() {

                return new Computer(this);

            }

        }

        public void displayConfiguration() {

            System.out.println("Computer Configuration:");

            System.out.println("CPU: " + CPU);

            System.out.println("RAM: " + RAM);

            System.out.println("Storage: " + storage);

            System.out.println();

        }

    }

    public static void main(String[] args) {

        Computer basicComputer = new Computer.Builder()

                .setCPU("Intel i3")

                .setRAM("4GB")

                .setStorage("256GB SSD")

                .build();

        Computer gamingComputer = new Computer.Builder()

                .setCPU("Intel i9")

                .setRAM("32GB")

                .setStorage("1TB SSD")

                .build();

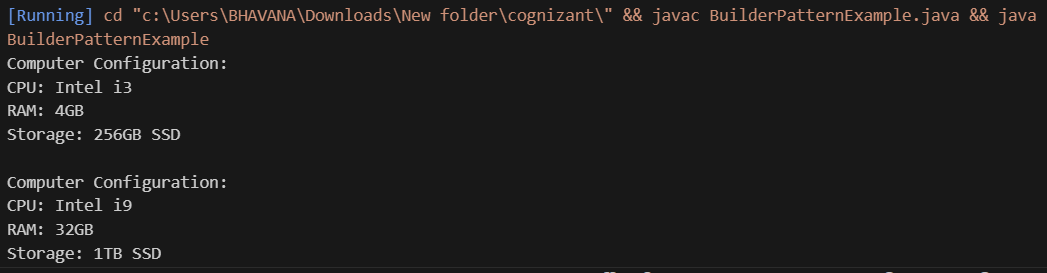
        basicComputer.displayConfiguration();

        gamingComputer.displayConfiguration();

    }

}

OUTPUT:



**Exercise 4: Implementing the Adapter Pattern**

public class AdapterPatternExample {

    interface PaymentProcessor {

        void processPayment(double amount);

    }

    static class PayPalGateway {

        public void sendPayment(double amountInDollars) {

            System.out.println("Processing payment via PayPal: $" + amountInDollars);

        }

    }

    static class StripeGateway {

        public void makeStripePayment(double amount) {

            System.out.println("Processing payment via Stripe: $" + amount);

        }

    }

    static class RazorpayGateway {

        public void doTransaction(double amount) {

            System.out.println("Processing payment via Razorpay: $" + amount);

        }

    }

    static class PayPalAdapter implements PaymentProcessor {

        private PayPalGateway payPalGateway;

        public PayPalAdapter() {

            this.payPalGateway = new PayPalGateway();

        }

        public void processPayment(double amount) {

            payPalGateway.sendPayment(amount);

        }

    }

    static class StripeAdapter implements PaymentProcessor {

        private StripeGateway stripeGateway;

        public StripeAdapter() {

            this.stripeGateway = new StripeGateway();

        }

        public void processPayment(double amount) {

            stripeGateway.makeStripePayment(amount);

        }

    }

    static class RazorpayAdapter implements PaymentProcessor {

        private RazorpayGateway razorpayGateway;

        public RazorpayAdapter() {

            this.razorpayGateway = new RazorpayGateway();

        }

        public void processPayment(double amount) {

            razorpayGateway.doTransaction(amount);

        }

    }

    public static void main(String[] args) {

        PaymentProcessor paypal = new PayPalAdapter();

        PaymentProcessor stripe = new StripeAdapter();

        PaymentProcessor razorpay = new RazorpayAdapter();

        paypal.processPayment(100.0);

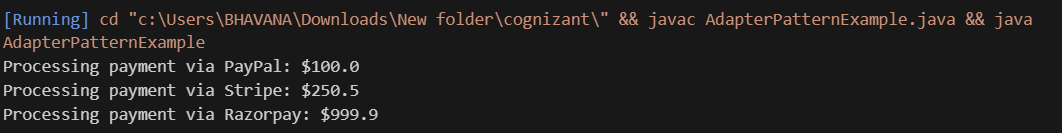
        stripe.processPayment(250.5);

        razorpay.processPayment(999.9);

    }

}

OUTPUT:



**Exercise 5: Implementing the Decorator Pattern**

public class DecoratorPatternExample {

    interface Notifier {

        void send(String message);

    }

    static class EmailNotifier implements Notifier {

        public void send(String message) {

            System.out.println("Sending Email: " + message);

        }

    }

    static abstract class NotifierDecorator implements Notifier {

        protected Notifier wrappedNotifier;

        public NotifierDecorator(Notifier notifier) {

            this.wrappedNotifier = notifier;

        }

        public void send(String message) {

            wrappedNotifier.send(message);

        }

    }

    static class SMSNotifierDecorator extends NotifierDecorator {

        public SMSNotifierDecorator(Notifier notifier) {

            super(notifier);

        }

        public void send(String message) {

            super.send(message);

            sendSMS(message);

        }

        private void sendSMS(String message) {

            System.out.println("Sending SMS: " + message);

        }

    }

    static class SlackNotifierDecorator extends NotifierDecorator {

        public SlackNotifierDecorator(Notifier notifier) {

            super(notifier);

        }

        public void send(String message) {

            super.send(message);

            sendSlack(message);

        }

        private void sendSlack(String message) {

            System.out.println("Sending Slack message: " + message);

        }

    }

    public static void main(String[] args) {

        Notifier baseNotifier = new EmailNotifier();

        Notifier multiChannelNotifier = new SlackNotifierDecorator(

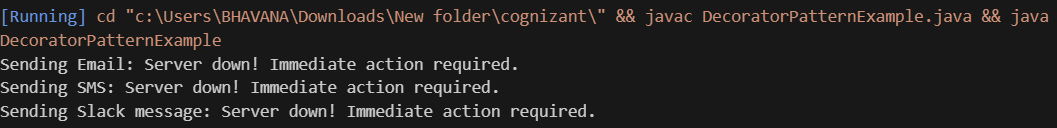
                                            new SMSNotifierDecorator(baseNotifier));

        multiChannelNotifier.send("Server down! Immediate action required.");

    }

}

OUTPUT:



**Exercise 6: Implementing the Proxy Pattern**

public class ProxyPatternExample {

    interface Image {

        void display();

    }

    static class RealImage implements Image {

        private String fileName;

        public RealImage(String fileName) {

            this.fileName = fileName;

            loadFromServer();

        }

        private void loadFromServer() {

            System.out.println("Loading image from remote server: " + fileName);

        }

        public void display() {

            System.out.println("Displaying image: " + fileName);

      }

    }

    static class ProxyImage implements Image {

        private String fileName;

        private RealImage realImage;

        public ProxyImage(String fileName) {

            this.fileName = fileName;

        }

        public void display() {

            if (realImage == null) {

                realImage = new RealImage(fileName);

            } else {

                System.out.println("Image already loaded, using cached version.");

            }

            realImage.display();

        }

    }

    public static void main(String[] args) {

        Image image1 = new ProxyImage("nature.jpg");

        image1.display();

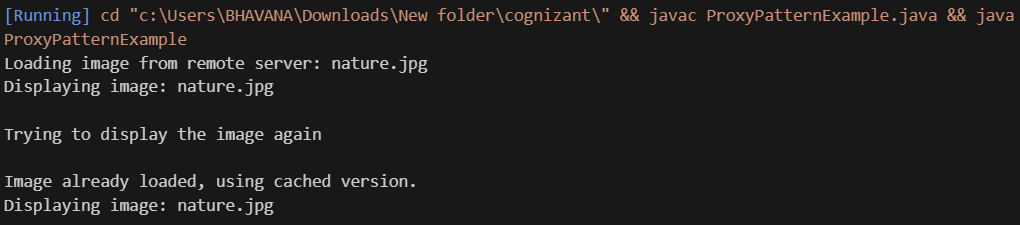
        System.out.println("\nTrying to display the image again\n");

        image1.display();

    }

}

OUTPUT:



**Exercise 7: Implementing the Observer Pattern**

import java.util.\*;

interface Stock {

    void registerObserver(Observer o);

    void removeObserver(Observer o);

    void notifyObservers();

}

interface Observer {

    void update(String stockName, double price);

}

class StockMarket implements Stock {

    private List<Observer> observers = new ArrayList<>();

    private String stockName;

    private double stockPrice;

    public void setStockPrice(String stockName, double price) {

        this.stockName = stockName;

        this.stockPrice = price;

        notifyObservers();

    }

    public void registerObserver(Observer o) {

        observers.add(o);

    }

    public void removeObserver(Observer o) {

        observers.remove(o);

    }

    public void notifyObservers() {

        for (Observer o : observers) {

            o.update(stockName, stockPrice);

        }

    }

}

class MobileApp implements Observer {

    private String appName;

    public MobileApp(String appName) {

        this.appName = appName;

    }

    public void update(String stockName, double price) {

        System.out.println(appName + " - Mobile Notification: " + stockName + " is now ₹" + price);

    }

}

class WebApp implements Observer {

    private String siteName;

    public WebApp(String siteName) {

        this.siteName = siteName;

    }

    public void update(String stockName, double price) {

        System.out.println(siteName + " - Web Alert: " + stockName + " stock updated to ₹" + price);

    }

}

public class ObserverPatternExample {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        Observer mobileApp1 = new MobileApp("StocksNow");

        Observer mobileApp2 = new MobileApp("InvestApp");

        Observer webApp = new WebApp("StockTracker.com");

        stockMarket.registerObserver(mobileApp1);

        stockMarket.registerObserver(mobileApp2);

        stockMarket.registerObserver(webApp);

        System.out.println("Updating stock price...\n");

        stockMarket.setStockPrice("INFY", 1475);

        System.out.println("\nRemoving one mobile app and updating again\n");

        stockMarket.removeObserver(mobileApp2);

        stockMarket.setStockPrice("TCS", 3500);

    }

}

OUTPUT:



**Exercise 8: Implementing the Strategy Pattern**

interface PaymentStrategy {

    void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

    private String cardNumber;

    public CreditCardPayment(String cardNumber) {

        this.cardNumber = cardNumber;

    }

    public void pay(double amount) {

        System.out.println("Paid ₹" + amount + " using Credit Card ending with " + cardNumber.substring(cardNumber.length() - 4));

    }

}

class PayPalPayment implements PaymentStrategy {

    private String email;

    public PayPalPayment(String email) {

        this.email = email;

    }

    public void pay(double amount) {

        System.out.println("Paid ₹" + amount + " using PayPal account: " + email);

    }

}

class PaymentContext {

    private PaymentStrategy strategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {

        this.strategy = strategy;

    }

    public void makePayment(double amount) {

        if (strategy == null) {

            System.out.println("No payment strategy selected.");

        } else {

            strategy.pay(amount);

        }

    }

}

public class StrategyPatternExample {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        context.setPaymentStrategy(new CreditCardPayment("1234567890123456"));

        context.makePayment(2500);

        System.out.println();

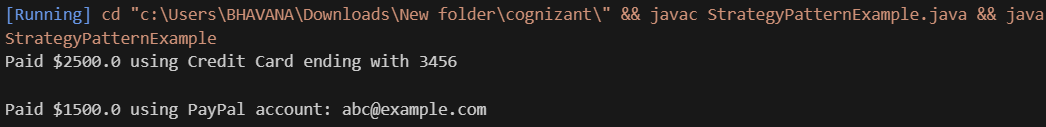
        context.setPaymentStrategy(new PayPalPayment("abc@example.com"));

        context.makePayment(1500);

    }

}

OUTPUT:



**Exercise 9: Implementing the Command Pattern**

interface Command {

    void execute();

}

class Light {

    public void turnOn() {

        System.out.println("Light is ON");

    }

    public void turnOff() {

        System.out.println("Light is OFF");

    }

}

class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }

    public void execute() {

        light.turnOn();

    }

}

class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    public void execute() {

        light.turnOff();

    }

}

class RemoteControl {

    private Command command;

    public void setCommand(Command command) {

        this.command = command;

    }

    public void pressButton() {

        if (command != null) {

            command.execute();

        } else {

            System.out.println("No command set!");

        }

    }

}

public class CommandPattern {

    public static void main(String[] args) {

        Light livingRoomLight = new Light();

        Command lightOn = new LightOnCommand(livingRoomLight);

        Command lightOff = new LightOffCommand(livingRoomLight);

        RemoteControl remote = new RemoteControl();

        System.out.println("Pressing ON button:");

        remote.setCommand(lightOn);

        remote.pressButton();

        System.out.println("\nPressing OFF button:");

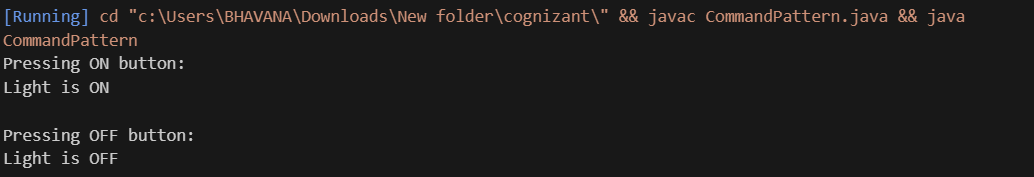
        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

OUTPUT:



**Exercise 10: Implementing the MVC Pattern**

class Student {

    private String name;

    private String id;

    private String grade;

    public Student(String name, String id, String grade) {

        this.name = name;

        this.id = id;

        this.grade = grade;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public String getId() {

        return id;

    }

    public String getGrade() {

        return grade;

    }

    public void setGrade(String grade) {

        this.grade = grade;

    }

}

class StudentView {

    public void displayStudentDetails(String name, String id, String grade) {

        System.out.println("Student Details");

        System.out.println("Name  : " + name);

        System.out.println("ID    : " + id);

        System.out.println("Grade : " + grade);

    }

}

class StudentController {

    private Student model;

    private StudentView view;

    public StudentController(Student model, StudentView view) {

        this.model = model;

        this.view = view;

    }

    public void setStudentName(String name) {

        model.setName(name);

    }

    public void setStudentGrade(String grade) {

        model.setGrade(grade);

    }

    public String getStudentName() {

        return model.getName();

    }

    public String getStudentId() {

        return model.getId();

    }

    public String getStudentGrade() {

        return model.getGrade();

    }

    public void updateView() {

        view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());

    }

}

public class MVCPatternExample {

    public static void main(String[] args) {

        Student student = new Student("abc", "101", "A");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.updateView();

        System.out.println("\nUpdating Student Grade\n");

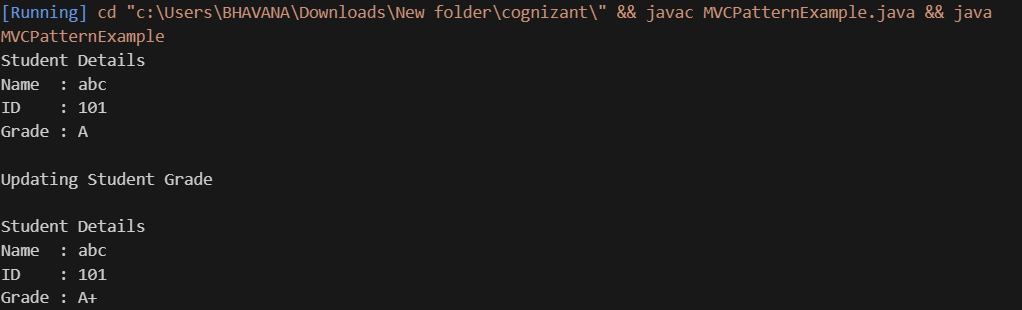
        controller.setStudentGrade("A+");

        controller.updateView();

    }

}

OUTPUT:



**Exercise 11: Implementing Dependency Injection**

interface CustomerRepository {

    String findCustomerById(String customerId);

}

class CustomerRepositoryImpl implements CustomerRepository {

    public String findCustomerById(String customerId) {

        return "Customer[ID=" + customerId + ", Name=abc, Location=Hyderabad]";

    }

}

class CustomerService {

    private CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public void showCustomer(String customerId) {

        String customer = customerRepository.findCustomerById(customerId);

        System.out.println("Retrieved: " + customer);

    }

}

public class DependencyInjectionExample {

    public static void main(String[] args) {

        CustomerRepository repository = new CustomerRepositoryImpl();

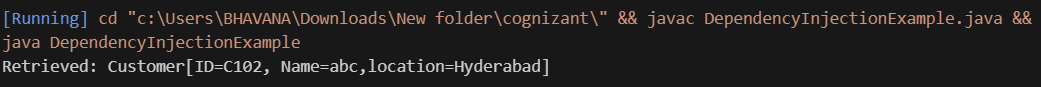
        CustomerService service = new CustomerService(repository);

        service.showCustomer("C102");

    }

}

OUTPUT:



**Module 2:Data Structures and Algorithms**

**Exercise 1: Inventory Management System**

Efficient data structures and algorithms are crucial because:

* A warehouse can store thousands of products, and each product operation (add, update, delete, search) must be fast and reliable.
* Poor choices can lead to slow searches, duplicate entries, and high memory usage.
* Efficient structures reduce the time complexity from O(n) to O(1) or O(log n).

|  |  |  |
| --- | --- | --- |
| Data Structure | Use Case | Advantages |
| ArrayList | Simple lists, when order matters | Easy iteration |
| HashMap | Quick lookup by product ID | O(1) average time for add, search, and delete operations |
| TreeMap | When sorted entries are needed | Maintains keys in sorted order |
| HashSet | Ensuring uniqueness of items | Prevents duplicate entries |

**Implementation:**

import java.util.HashMap;

import java.util.Scanner;

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;

}

@Override

public String toString() {

return "[" + productId + "] " + productName + " | Qty: " + quantity + " | Price: Rs." + price;

}

}

public class InventoryManagementSystem {

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

public void addProduct(Product p) {

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

System.out.println(“Product already exists. Use updateProduct.");

} else {

inventory.put(p.productId, p);

System.out.println("Product added.");

}

}

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

if (inventory.containsKey(productId)) {

Product p = inventory.get(productId);

p.quantity = quantity;

p.price = price;

System.out.println(“Product updated");

} else {

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

}

}

public void deleteProduct(String productId) {

if (inventory.containsKey(productId)) {

inventory.remove(productId);

System.out.println("Product deleted.");

} else {

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

}

}

public void showInventory() {

if (inventory.isEmpty()) {

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

} else {

System.out.println("\nCurrent Inventory:");

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

System.out.println(p);

}

}

}

public static void main(String[] args) {

InventoryManagementSystem ims = new InventoryManagementSystem();

Scanner scanner = new Scanner(System.in);

int choice;

do {

System.out.println("\n Inventory Management Menu");

System.out.println("1. Add Product");

System.out.println("2. Update Product");

System.out.println("3. Delete Product");

System.out.println("4. Show Inventory");

System.out.println("0. Exit");

System.out.print("Enter your choice: ");

choice = scanner.nextInt();

scanner.nextLine();

String productId, productName;

int quantity;

double price;

switch (choice) {

case 1:

System.out.print("Enter Product ID: ");

productId = scanner.nextLine();

System.out.print("Enter Product Name: ");

productName = scanner.nextLine();

System.out.print("Enter Quantity: ");

quantity = scanner.nextInt();

System.out.print("Enter Price: ");

price = scanner.nextDouble();

ims.addProduct(new Product(productId, productName, quantity, price));

break;

case 2:

System.out.print("Enter Product ID to update: ");

productId = scanner.nextLine();

System.out.print("Enter New Quantity: ");

quantity = scanner.nextInt();

System.out.print("Enter New Price: ");

price = scanner.nextDouble();

ims.updateProduct(productId, quantity, price);

break;

case 3:

System.out.print("Enter Product ID to delete: ");

productId = scanner.nextLine();

ims.deleteProduct(productId);

break;

case 4:

ims.showInventory();

break;

case 0:

System.out.println(“Exiting...Thank you!");

break;

default:

System.out.println("Invalid choice. Try again.");

}

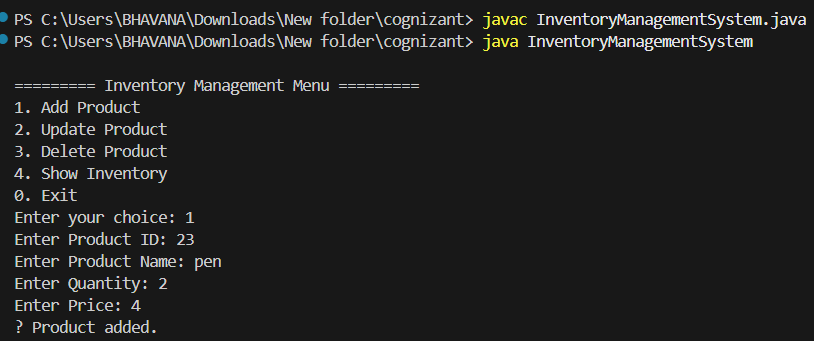
} while (choice != 0);

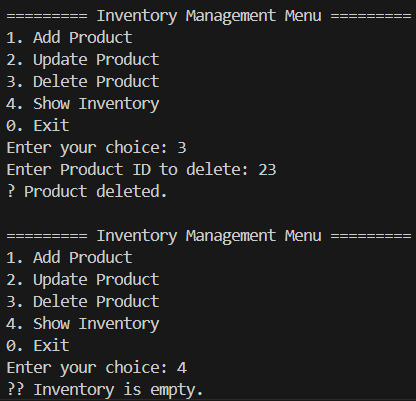
scanner.close();

}

}

OUTPUT:





Inventory Management System using a HashMap:

* The addProduct operation has a time complexity of O(1) on average. This is because HashMap uses hashing to directly index the location where the product should be stored. However, in the worst-case scenario, such as when many hash collisions occur, the complexity could degrade to O(n).
* The updateProduct operation also runs in O(1) time on average. Since the product ID is used as the key, accessing and modifying the associated product happens in constant time.
* The deleteProduct operation likewise takes O(1) time on average. Deletion by key in a HashMap is efficient and doesn't require shifting elements like in an array.
* The showInventory operation has a time complexity of O(n), where n is the number of products in the inventory. This is because it must iterate over all values in the HashMap to display each product.

To make your system work better, don’t use keys that can change, because it makes finding data harder. If your system is big, use a database or save data in files instead of keeping everything in memory. Also, if you often search by product name or price, adding indexes can help you find things faster.

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

#### **Big O Notation**:

Big O notation describes how the runtime of an algorithm grows as the input size increases. It focuses on the **worst-case scenario** and helps compare the **efficiency** of different algorithms, especially as inputs grow large.

· **O(1)** – Constant time

· **O(log n)** – Logarithmic time

· **O(n)** – Linear time

· **O(n log n)** – Log-linear time

· **O(n²)** – Quadratic time

|  |  |  |
| --- | --- | --- |
| Case Type | Linear Search | Binary Search |
| Best | O(1) | O(1) |
| Average | O(n) | O(log n) |
| Worst | O(n) | O(log n) |

**Linear Search s**cans each item; worst case is when the item is at the end or not present.

**Binary Search o**nly works on sorted arrays; it halves the search space at each step.

**Linear Search:**

import java.util.Scanner;

public class LinearSearchDemo {

public static Product linearSearch(Product[] products, String target) {

for (Product product : products) {

if (product.productName.equalsIgnoreCase(target)) {

return product;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shirt", "Clothing"),

new Product(3, "Pen", "Stationery"),

new Product(4, "Phone", "Electronics")

};

Scanner sc = new Scanner(System.in);

System.out.print("Enter product name to search (Linear Search): ");

String input = sc.nextLine();

Product found = linearSearch(products, input);

if (found != null) {

System.out.println("Product found: " + found);

} else {

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

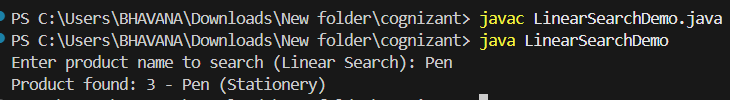
}

sc.close();

}

}

OUTPUT:



**Binary Search:**

import java.util.\*;

public class BinarySearchDemo {

public static Product binarySearch(Product[] products, String target) {

Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

int left = 0, right = products.length - 1;

while (left <= right) {

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

int cmp = products[mid].productName.compareToIgnoreCase(target);

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

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

else right = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shirt", "Clothing"),

new Product(3, "Pen", "Stationery"),

new Product(4, "Phone", "Electronics")

};

Scanner sc = new Scanner(System.in);

System.out.print("Enter product name to search: ");

String input = sc.nextLine();

Product found = binarySearch(products, input);

if (found != null) {

System.out.println("Product found: " + found);

} else {

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

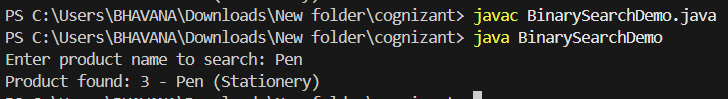
}

sc.close();

}

}

OUTPUT:



**Analysis:**

|  |  |  |
| --- | --- | --- |
| Algorithm | Time Complexity | Space Complexity |
| Linear Search | O(n) | O(1) |
| Binary Search | O(log n) | O(1) |

· **Linear Search** is simple and works on **unsorted** data, but is slow for large datasets.

· **Binary Search** is much faster but requires the array to be **sorted**, which may add overhead if data changes frequently.

**Conclusion**:

**Binary Search** is preferred if the product list is **pre-sorted or static**. It’s much faster and better for large datasets.Using Visual Studio, binary search makes the solution efficient and production-ready.

**Exercise 3: Sorting Customer Orders**

**Understand Sorting Algorithms:**

**Bubble Sort**

* Repeatedly compares adjacent elements and swaps them if they are in the wrong order.
* **Time Complexity**:

Best: O(n) (when already sorted)

Average/Worst: O(n²)

**Insertion Sort**

* Builds the final sorted array one element at a time by comparing it backward.
* **Time Complexity**:

Best: O(n)

Average/Worst: O(n²)

**Quick Sort**

* Picks a pivot, partitions elements into two halves (less and greater than the pivot), and sorts them recursively.
* **Time Complexity**:

Best/Average: O(n log n)

Worst: O(n²) (rare, when pivot is poorly chosen)

**Merge Sort**

* Divides the array into halves, sorts each half, and then merges them.
* **Time Complexity**:

Always O(n log n)

Extra space is required.

**Using Bubble Sort:**

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;

}

@Override

public String toString() {

return "[" + orderId + "] " + customerName + " - ₹" + totalPrice;

}

}

public class BubbleSortOrders {

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].totalPrice > orders[j + 1].totalPrice) {

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

public static void main(String[] args) {

Order[] orders = {

new Order("O001", "Alice", 2500.0),

new Order("O002", "Bob", 1800.0),

new Order("O003", "Charlie", 3200.0),

new Order("O004", "Diana", 1000.0)

};

System.out.println("Before Bubble Sort:");

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

bubbleSort(orders);

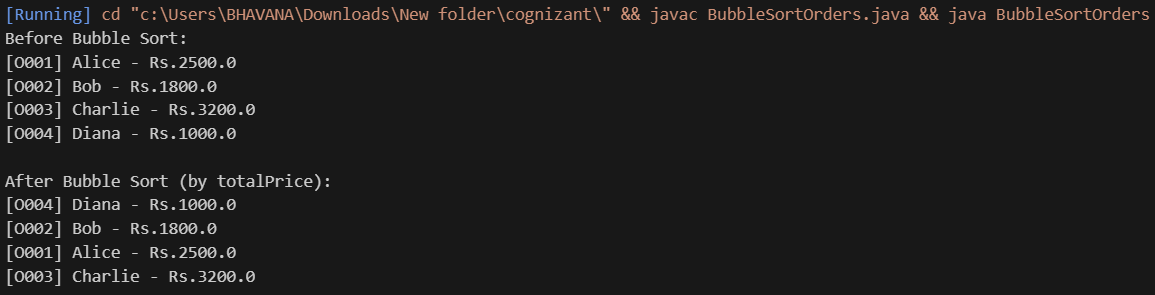
System.out.println("\nAfter Bubble Sort (by totalPrice):");

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

}

}

OUTPUT:



Analysis:

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Best Case | Average Case | Worst Case |
| Bubble Sort | O(n) | O(n²) | O(n²) |
| Quick Sort | O(n log n) | O(n log n) | O(n²) |

Bubble Sort is not suitable for large lists because it makes many unnecessary comparisons and swaps, which slows down the process. Whereas, Quick Sort is much faster, especially when the data is randomly arranged. It uses a divide-and-conquer approach to break the problem into smaller parts, which helps reduce sorting time. This makes Quick Sort a better choice for real-time and large-scale applications, such as e-commerce platforms where quick data processing is important.

**Exercise 4: Employee Management System**

* Arrays are contiguous blocks of memory.
* The base address of the array is the address of the first element.
* You can access any element directly using its index as arr[i], which is computed as base\_address + i \* size\_of\_element.

**Advantages of Arrays:**

Arrays offer fast access to elements, allowing you to retrieve any item in constant time (O(1)) using its index. They are simple to use, easy to declare, and straightforward to iterate through. Additionally, arrays use memory efficiently since they don't have the extra overhead of pointers that linked lists require.

**Implementation**:

class Employee {

int employeeId;

String name;

String position;

double salary;

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

this.employeeId = employeeId;

this.name = name;

this.position = position;

this.salary = salary;

}

@Override

public String toString() {

return "[" + employeeId + "] " + name + " | " + position + " | ₹" + salary;

}

}

public class EmployeeManagementSystem {

private Employee[] employees;

private int count;

public EmployeeManagementSystem(int size) {

employees = new Employee[size];

count = 0;

}

public void addEmployee(Employee emp) {

if (count < employees.length) {

employees[count++] = emp;

System.out.println("Employee added.");

} else {

System.out.println("Array is full. Cannot add more employees.");

}

}

public Employee searchEmployee(int id) {

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

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

return employees[i];

}

}

return null;

}

public void showAllEmployees() {

if (count == 0) {

System.out.println("No employees found.");

return;

}

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

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

}

}

public void deleteEmployee(int id) {

int index = -1;

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

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

index = i;

break;

}

}

if (index != -1) {

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

employees[i] = employees[i + 1];

}

employees[--count] = null;

System.out.println("Employee deleted.");

} else {

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

}

}

public static void main(String[] args) {

EmployeeManagementSystem ems = new EmployeeManagementSystem(5);

ems.addEmployee(new Employee(101, "Alice", "Manager", 70000));

ems.addEmployee(new Employee(102, "Bob", "Developer", 55000));

ems.addEmployee(new Employee(103, "Charlie", "Designer", 50000));

System.out.println("\nAll Employees:");

ems.showAllEmployees();

System.out.println("\nSearching for Employee with ID 102:");

Employee found = ems.searchEmployee(102);

if (found != null)

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

else

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

System.out.println("\nDeleting Employee with ID 101:");

ems.deleteEmployee(101);

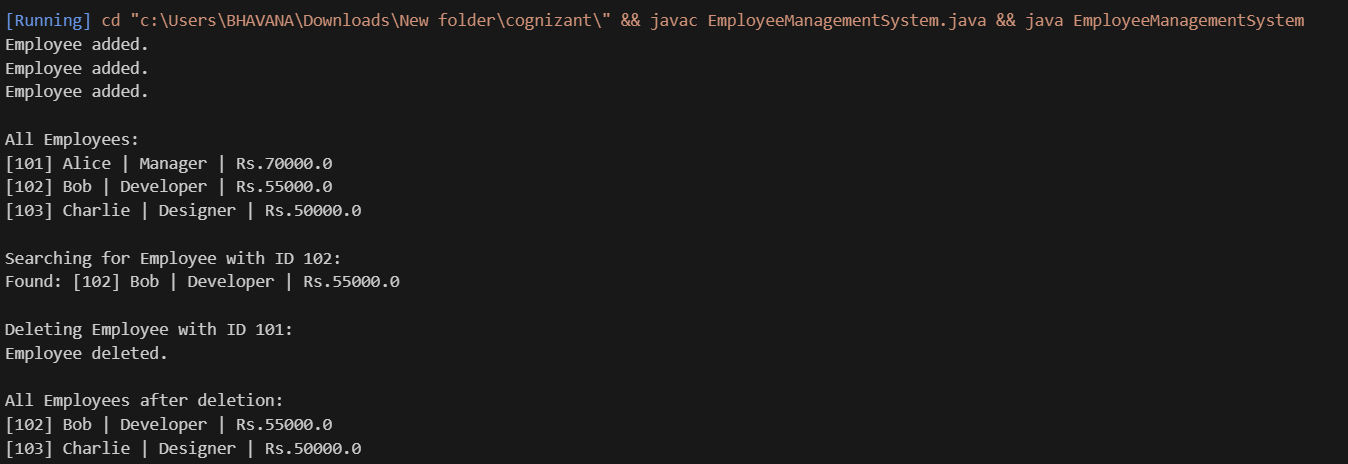
System.out.println("\nAll Employees after deletion:");

ems.showAllEmployees();

}

}

OUTPUT:



**Analysis**:

### **Limitations of Arrays:**

Arrays have a fixed size, which means they cannot grow beyond the capacity defined at the time of creation. Deleting or inserting elements can be costly because it often requires shifting other elements to maintain order. Additionally, if the array size is overestimated, it can lead to inefficient memory usage due to unused space.

### **When to Use Arrays:**

Arrays are used when the number of elements is known and fixed. They are ideal for situations where fast access to elements using an index is important. Arrays also work well for simple and static data structures that don't require frequent resizing or dynamic updates.

|  |  |
| --- | --- |
| Operation | Time Complexity |
| Add | O(1) |
| Search | O(n) |
| Traverse | O(n) |
| Delete | O(n) |

**Exercise 5: Task Management System**

### **1. Understanding Linked Lists**

**Singly Linked List:**

* Each node contains data and a reference to the next node.
* It can only be traversed in one direction (forward).

**Doubly Linked List:**

* Each node contains data, a reference to the next node, and a reference to the previous node.
* It allows traversal in both directions.
* Requires more memory (extra pointer for previous node).

**Implementation**:

class Task {

int taskId;

String taskName;

String status;

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

this.taskId = taskId;

this.taskName = taskName;

this.status = status;

}

public void display() {

System.out.println("ID: " + taskId + ", Name: " + taskName + ", Status: " + status);

}

}

class TaskNode {

Task task;

TaskNode next;

public TaskNode(Task task) {

this.task = task;

this.next = null;

}

}

public class TaskManagementSystem {

private TaskNode head;

public void addTask(Task task) {

TaskNode newNode = new TaskNode(task);

if (head == null) {

head = newNode;

} else {

TaskNode temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

System.out.println("Task added: " + task.taskName);

}

public Task searchTask(int taskId) {

TaskNode temp = head;

while (temp != null) {

if (temp.task.taskId == taskId) {

return temp.task;

}

temp = temp.next;

}

return null;

}

public void traverseTasks() {

TaskNode temp = head;

if (temp == null) {

System.out.println("No tasks to display.");

return;

}

System.out.println("Task List:");

while (temp != null) {

temp.task.display();

temp = temp.next;

}

}

public void deleteTask(int taskId) {

if (head == null) {

System.out.println("List is empty.");

return;

}

if (head.task.taskId == taskId) {

head = head.next;

System.out.println("Task with ID " + taskId + " deleted.");

return;

}

TaskNode current = head;

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

current = current.next;

}

if (current.next != null) {

current.next = current.next.next;

System.out.println("Task with ID " + taskId + " deleted.");

} else {

System.out.println("Task with ID " + taskId + " not found.");

}

}

public static void main(String[] args) {

TaskManagementSystem tms = new TaskManagementSystem();

tms.addTask(new Task(1, "Design Homepage", "Pending"));

tms.addTask(new Task(2, "Setup Database", "In Progress"));

tms.addTask(new Task(3, "Deploy Project", "Not Started"));

tms.traverseTasks();

System.out.println("\nSearching for Task ID 2:");

Task foundTask = tms.searchTask(2);

if (foundTask != null) {

foundTask.display();

} else {

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

}

System.out.println("\nDeleting Task ID 1:");

tms.deleteTask(1);

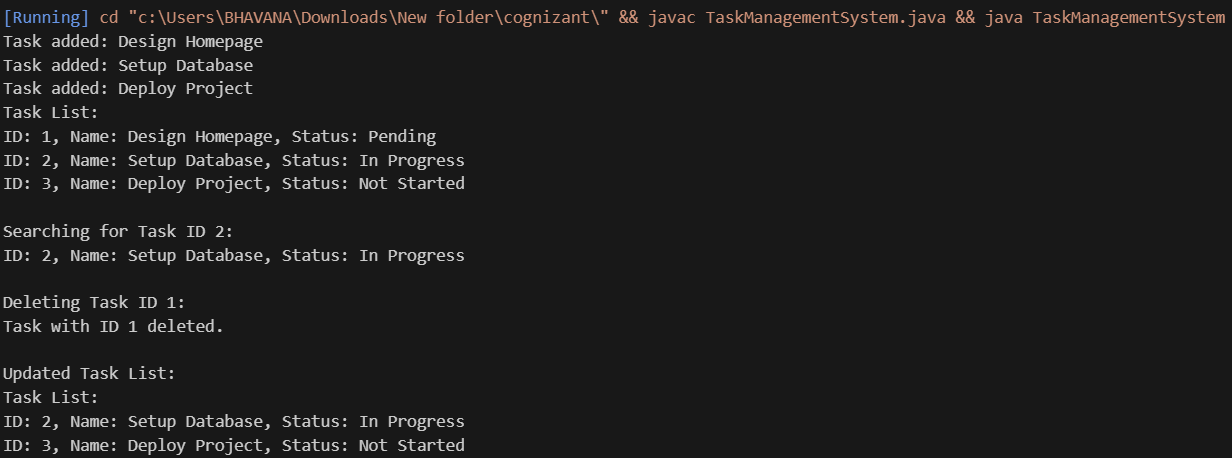
System.out.println("\nUpdated Task List:");

tms.traverseTasks();

}

}

OUTPUT:



**Analysis**:

|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Explanation |
| Add | O(n) | Traverse to the end to insert node. |
| Search | O(n) | Linear traversal through the list. |
| Traverse | O(n) | Visit each node one by one. |
| Delete | O(n) | Find the node before deletion point. |

· Linked lists are ideal when the size of the dataset changes frequently.

· Insertion and deletion at the beginning or middle are much faster than arrays.

· Linked lists use memory only as needed, reducing wasted space.

**Exercise 6: Library Management System**

### **1. Understanding Search Algorithms**

#### **Linear Search**

* Traverse the list one element at a time.
* No need for the data to be sorted.

**Time Complexity**:

* Best: O(1)
* Average/Worst: O(n)

#### ****Binary Search****

* Works only on **sorted lists**.
* Repeatedly divide the list in half to locate the target.

**Time Complexity**:

* Best: O(1)
* Average/Worst: O(log n)

**Implementation:**

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 void display() {

System.out.println("ID: " + bookId + ", Title: " + title + ", Author: " + author);

}

}

public class LibraryManagementSystem {

Book[] books;

int size;

public LibraryManagementSystem(int capacity) {

books = new Book[capacity];

size = 0;

}

public void addBook(Book book) {

if (size < books.length) {

books[size++] = book;

System.out.println("Book added: " + book.title);

} else {

System.out.println("Library is full.");

}

}

public Book linearSearch(String title) {

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

if (books[i].title.equalsIgnoreCase(title)) {

return books[i];

}

}

return null;

}

public Book binarySearch(String title) {

Arrays.sort(books, 0, size, Comparator.comparing(book -> book.title.toLowerCase()));

int low = 0, high = size - 1;

while (low <= high) {

int mid = (low + high) / 2;

int compare = books[mid].title.compareToIgnoreCase(title);

if (compare == 0) {

return books[mid];

} else if (compare < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

public void displayAllBooks() {

if (size == 0) {

System.out.println("No books in the library.");

} else {

System.out.println("Library Book List:");

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

books[i].display();

}

}

}

public static void main(String[] args) {

LibraryManagementSystem library = new LibraryManagementSystem(10);

library.addBook(new Book(1, "The Alchemist", "Paulo Coelho"));

library.addBook(new Book(2, "Wings of Fire", "A.P.J. Abdul Kalam"));

library.addBook(new Book(3, "Rich Dad Poor Dad", "Robert Kiyosaki"));

System.out.println();

library.displayAllBooks();

System.out.println("\nLinear Search for 'Wings of Fire':");

Book foundLinear = library.linearSearch("Wings of Fire");

if (foundLinear != null) foundLinear.display();

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

System.out.println("\nBinary Search for 'Rich Dad Poor Dad':");

Book foundBinary = library.binarySearch("Rich Dad Poor Dad");

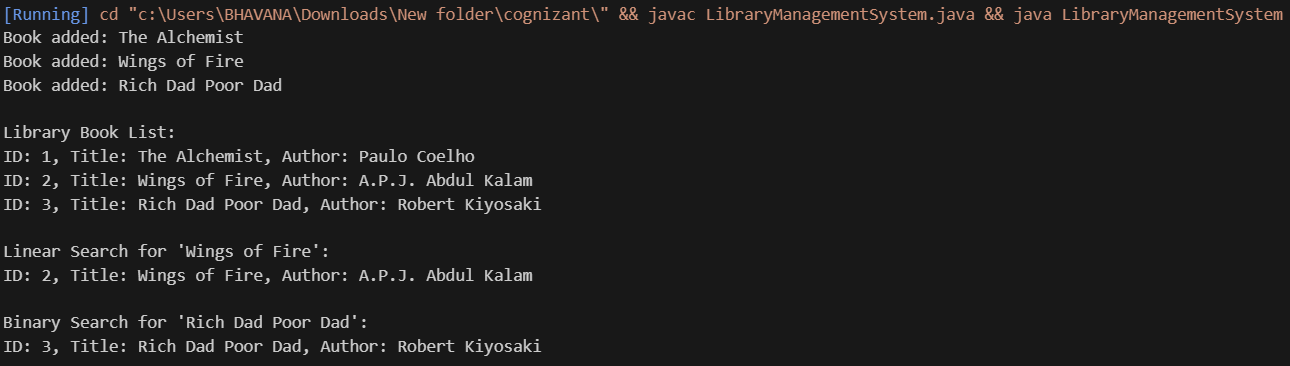
if (foundBinary != null) foundBinary.display();

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

}

}

OUTPUT:



**Analysis:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Search Type | Best Case | Average Case | Worst Case | Sorted Required |
| Linear Search | O(1) | O(n) | O(n) | No |
| Binary Search | O(1) | O(log n) | O(log n) | Yes |

|  |  |
| --- | --- |
| Condition | Recommended Algorithm |
| Unsorted or small dataset | Linear Search |
| Large and sorted dataset | Binary Search |
| Dataset changes frequently (insertions) | Linear Search |
| Dataset is static | Binary Search |

**Exercise 7: Financial Forecasting**

**Recursion** is a technique where a function calls itself to solve smaller sub-problems of the original problem.It helps simplify problems like factorial and Fibonacci series.

**Implementation:**

public class FinancialForecast {

public static double futureValue(double initialValue, double growthRate, int years) {

if (years == 0) {

return initialValue;

}

return futureValue(initialValue, growthRate, years - 1) \* (1 + growthRate);

}

public static void main(String[] args) {

explainRecursion();

double initialValue = 1000.0;

double growthRate = 0.10;

int years = 5;

System.out.println("Forecasting for " + years + " years with 10% growth..");

double resultRecursive = futureValue(initialValue, growthRate, years);

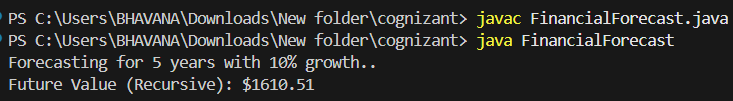
System.out.printf("Future Value (Recursive): $%.2f%n", resultRecursive);

System.out.println();

}

}

OUTPUT:



Time complexity of this recursive algorithm is O(n).This can be optimized to avoid excessive computation using iteration:

public class FinancialForecast {

public static double futureValueOptimized(double initialValue, double growthRate, int years) {

double result = initialValue;

for (int i = 1; i <= years; i++) {

result \*= (1 + growthRate);

}

return result;

}

public static void main(String[] args) {

double initialValue = 1000.0;

double growthRate = 0.10;

int years = 5;

double resultOptimized = futureValueOptimized(initialValue, growthRate, years);

System.out.printf("Future Value (Optimized): $%.2f%n", resultOptimized);

System.out.println();

}

}

OUTPUT:

