**Data Structures And Algorithm  
Mandatory**

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

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

**Steps:**

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

**Solution**

**File: Product.java**

public class Product {

private int id;

private String name;

private int qty;

private double price;

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

this.id = id;

this.name = name;

this.qty = qty;

this.price = price;

}

public int getId() { return id; }

public String getName() { return name; }

public int getQty() { return qty; }

public double getPrice() { return price; }

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

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

public String toString() {

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

}

}

**File: Inventory.java**

import java.util.\*;

public class Inventory {

private HashMap<Integer, Product> products;

public Inventory() {

products = new HashMap<>();

}

public void add(Product p) {

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

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

}

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

Product p = products.get(id);

if (p != null) {

p.setQty(newQty);

p.setPrice(newPrice);

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

} else {

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

}

}

public void delete(int id) {

Product removed = products.remove(id);

if (removed != null) {

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

} else {

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

}

}

public void display() {

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

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

System.out.println(p);

}

}

}

**File: InventoryTest.java**

public class InventoryTest {

public static void main(String[] args) {

Inventory inv = new Inventory();

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

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

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

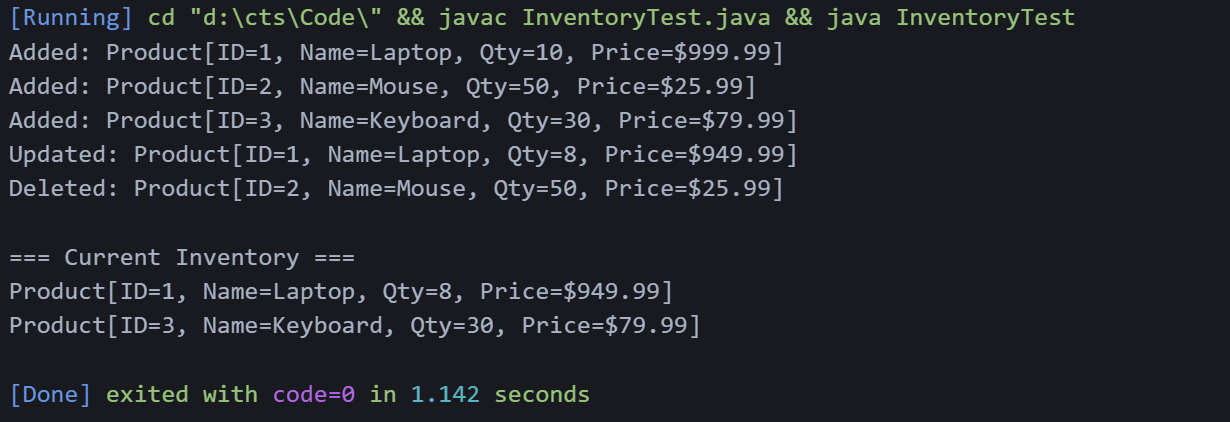
inv.update(1, 8, 949.99);

inv.delete(2);

inv.display();

}

}

**Output:  
**

**Analysis:  
Linear Search:  
Best Case: O(1)  
Average Case:O(n)**

**Worst Case:O(n)**

**Binary Search:  
Best Case: O(1)  
Average Case:O(log n)**

**Worst Case:O(log n)**

**Conclusion:  
So,binary Search is best.**

**7)Financial Forecasting**

**Scenario:**

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

**Steps:**

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

**Solution**

**FinancialForecasting.java**

public class FinancialForecasting {

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

if (years == 0) {

return presentValue;

}

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

}

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

double futureValue = presentValue;

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

futureValue \*= (1 + growthRate);

}

return futureValue;

}

}

**ForecastingTest.java**

public class ForecastingTest {

public static void main(String[] args) {

double presentValue = 1000.0;

double growthRate = 0.05;

int years = 10;

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

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

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

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

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

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

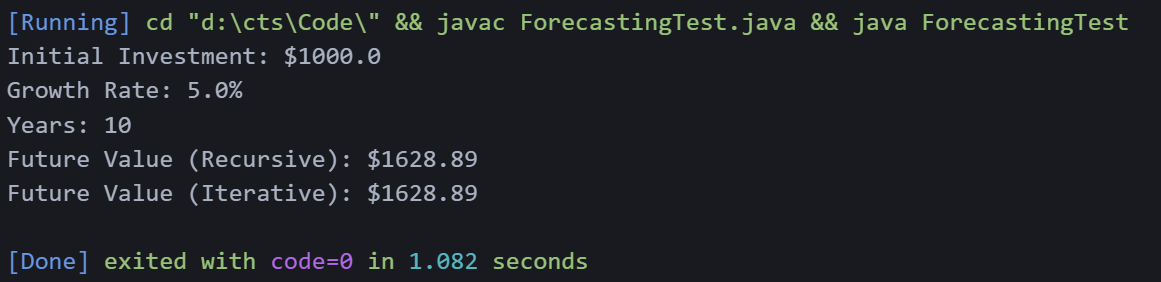
System.out.println("Future Value (Iterative): $" + String.format("%.2f", iterativeResult));

}

}

**Output:**

**Screenshots:**

****

**Analysis:**

**Time Complexity:  
Best Case:O(1)  
Average Case:O(n)  
Worst Case:O(n)**

**Optimization Strategies**

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