* **Algorithm\_Data Structure**

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

**Step 1: Understand Asymptotic Notation**

**🔹 Big O Notation**

**Big O notation describes how the runtime or space requirements of an algorithm grow** relative to the input size (n).

* **O(1)**: Constant time
* **O(n)**: Linear time
* **O(log n)**: Logarithmic time
* **O(n log n)**: Linearithmic time

**🔹 Best, Average, Worst Case**

For **search operations**:

* **Linear Search**:
  + Best: O(1) (element is first)
  + Average: O(n/2) ≈ O(n)
  + Worst: O(n)
* **Binary Search** (on a sorted array):
  + Best: O(1)
  + Average/Worst: O(log n)

**Step 2 and 3: Setup- Product Class and Implementation- Linear and Binary Search**

import java.util.Arrays;

import java.util.Comparator;

public class ECommerceSearch {

// Product class

static class Product {

int productId;

String productName;

String category;

Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

@Override

public String toString() {

return "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

// Linear Search

public static Product linearSearch(Product[] products, int targetId) {

for (Product p : products) {

if (p.productId == targetId) {

return p;

}

}

return null;

}

// Binary Search (on sorted array)

public static Product binarySearch(Product[] products, int targetId) {

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

if (products[mid].productId == targetId) {

return products[mid];

} else if (products[mid].productId < targetId) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

// Main method - program entry point

public static void main(String[] args) {

Product[] products = {

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

new Product(101, "Shoes", "Footwear"),

new Product(103, "Laptop", "Electronics"),

new Product(102, "T-shirt", "Apparel")

};

int targetId = 103;

// Linear Search

Product resultLinear = linearSearch(products, targetId);

System.out.println("Linear Search Result: " + (resultLinear != null ? resultLinear : "Product Not Found"));

// Sort for Binary Search

Arrays.sort(products, Comparator.comparingInt(p -> p.productId));

// Binary Search

Product resultBinary = binarySearch(products, targetId);

System.out.println("Binary Search Result: " + (resultBinary != null ? resultBinary : "Product Not Found"));

}

}

**OUTPUT:**

A computer screen shot of a computer screen

AI-generated content may be incorrect.

**Step 4: Analysis**

**Time Complexity:**

| **Algorithm** | **Time Complexity** | **Sorted Needed?** |
| --- | --- | --- |
| Linear Search | O(n) | No |
| Binary Search | O(log n) | Yes |

**Which is better?**

* **Binary Search** is better for **large, sorted datasets**, ideal for a performance-optimized e-commerce platform.
* If data is dynamic and unsorted, **linear search** is simpler but **slower**.
* In practice, **HashMaps or indexing (like in databases or Elasticsearch)** are even faster (O(1) average for lookup).

**Exercise 7: Financial Forecasting**

Java Code -Financial Forecasting Using Recursion

import java.util.HashMap;

public class FinancialForecast {

// Recursive function to calculate future value

// Formula: futureValue = presentValue \* (1 + growthRate)^years

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

if (years == 0) return presentValue;

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

}

// Optimized recursive method with memoization

static HashMap<Integer, Double> memo = new HashMap<>();

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

if (years == 0) return presentValue;

if (memo.containsKey(years)) return memo.get(years);

double result = (1 + growthRate) \* forecastValueMemoized(presentValue, growthRate, years - 1);

memo.put(years, result);

return result;

}

// Main method for testing

public static void main(String[] args) {

double presentValue = 1000.0; // Initial value

double growthRate = 0.10; // 10% annual growth

int years = 5;

System.out.println("Recursive Forecast Value after " + years + " years: ₹" +

forecastValueRecursive(presentValue, growthRate, years));

// Clear memoization map

memo.clear();

System.out.println("Memoized Forecast Value after " + years + " years: ₹" +

forecastValueMemoized(presentValue, growthRate, years));

}

}

**OUTPUT:**

A computer screen shot of a computer screen

AI-generated content may be incorrect.

**Step 1: Recursion Concept:**

* A function calling itself to solve smaller instances of the same problem.
* Simplifies problems like growth prediction where future depends on past.

**Step 2:Forecast Formula:**

* Repeats: future = present \* (1 + r)^n
* Implemented recursively as:

forecast(pv, r, n) = (1 + r) \* forecast(pv, r, n-1)

**Step 3:Time Complexity:**

* **Naive Recursion**: O(n)
* **Memoized Recursion**: O(n), but with reduced **repeated computation** thanks to caching.
* **Design Principles and Pattern**

**Exercise 2: Implementating the Singleton Pattern**

**Step 1 and 2:** **Singleton Class and Test Class to Verify Singleton**

Code:

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("Application started.");

logger2.log("User logged in.");

if (logger1 == logger2) {

System.out.println("Only one instance of Logger is used (Singleton confirmed).");

} else {

System.out.println("Different instances exist (Singleton failed).");

}

}

}

// Non-public Logger class inside same file

class Logger {

private static Logger instance;

private Logger() {

System.out.println("Logger instance created.");

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**Singleton Pattern in Java**

**Definition:**

The **Singleton Pattern** ensures that a **class has only one instance** throughout the application and provides a **global point of access** to that instance.

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

1.Define Document Interface

// IDocument.java

public interface IDocument {

void open();

}

2. Create Concrete Document Classes

// WordDocument.java

public class WordDocument implements IDocument {

public void open() {

System.out.println("Opening Word Document...");

}

}

// PdfDocument.java

public class PdfDocument implements IDocument {

public void open() {

System.out.println("Opening PDF Document...");

}

}

// ExcelDocument.java

public class ExcelDocument implements IDocument {

public void open() {

System.out.println("Opening Excel Document...");

}

}

3. Create Abstract Factory Class

// DocumentFactory.java

public abstract class DocumentFactory {

public abstract IDocument createDocument();

}

4. Create Concrete Factory Classes

// WordDocumentFactory.java

public class WordDocumentFactory extends DocumentFactory {

public IDocument createDocument() {

return new WordDocument();

}

}

// PdfDocumentFactory.java

public class PdfDocumentFactory extends DocumentFactory {

public IDocument createDocument() {

return new PdfDocument();

}

}

// ExcelDocumentFactory.java

public class ExcelDocumentFactory extends DocumentFactory {

public IDocument createDocument() {

return new ExcelDocument();

}

}

5. Test the Factory Method Pattern

// Main.java

public class Main {

public static void main(String[] args) {

// Create factories

DocumentFactory wordFactory = new WordDocumentFactory();

DocumentFactory pdfFactory = new PdfDocumentFactory();

DocumentFactory excelFactory = new ExcelDocumentFactory();

// Create and open documents

IDocument wordDoc = wordFactory.createDocument();

wordDoc.open();

IDocument pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

IDocument excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

OUTPUT:

A screenshot of a computer

AI-generated content may be incorrect.