Superset ID: 6373431

COGNIZANT DIGITAL NURTURE 4.0 JAVA FSE

WEEK-1

ALGORITHMS\_DATA STRUCTURES

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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**1. Understand Asymptotic Notation**

Big O Notation:

Big O notation is used to describe the efficiency of an algorithm, specifically:

* How the algorithm’s time or space requirements grow with input size.
* It helps in comparing different algorithms regardless of hardware.

Best, Average, and Worst-Case Scenarios

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

* Best Case: Element is found early (or in the middle for binary).
* Average Case: Element is somewhere in the middle.
* Worst Case: Element is at the end (linear) or not present (both).

**2. Setup**

class Product {

int productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

}

**3. Implementation**

Linear Search

public class ProductSearch {

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

for (Product product : products) {

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

return product;

}

}

return null;

}

Binary Search (sorted by productName)

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

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

while (low <= high) {

int mid = (low + high) / 2;

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

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

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

else high = mid - 1;

}

return null;

}

}

**4. Analysis**

Time Complexity Comparison

|  |  |  |  |
| --- | --- | --- | --- |
| Search Type | Time Complexity | Space Complexity | Sorted Required |
| Linear Search | O(n) | O(1) | No |
| Binary Search | O(log n) | O(1) | Yes |

Let’s say we have 1 million products:

* Linear Search:
  + May scan up to 1,000,000 comparisons in the worst case.
  + Slower response times for users.
* Binary Search:
  + Only needs around log₂(1,000,000) ≈ 20 comparisons.
  + Much faster and scalable.

Which is More Suitable?

* Use linear search when:
  + The dataset is small.
  + Data isn’t sorted or sorting isn’t feasible.
* Use binary search when:
  + The dataset is large and sorted.
  + Performance is critical for frequent lookups.
* For a simple search system, **binary search** is clearly more efficient **if data is sorted**.
* In practical large-scale applications, binary search principles form the foundation, but are enhanced with **indexing, ranking algorithms, and AI-powered search models**.

CODE

Product.java

package com.example.search;

class Product {

int productId;

String productName;

String category;

public 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;

}

}

ProductSearch.java

package com.example.search;

import java.util.Arrays;

import java.util.Comparator;

public class ProductSearch {

// Linear Search

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

for (Product product : products) {

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

return product;

}

}

return null;

}

// Binary Search (Assumes array is sorted by productName)

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

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

while (low <= high) {

int mid = (low + high) / 2;

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

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

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

else high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(101, "Laptop", "Electronics"),

new Product(102, "Shirt", "Apparel"),

new Product(103, "Coffee Maker", "Kitchen"),

new Product(104, "Book", "Education"),

new Product(105, "Headphones", "Electronics")

};

// Linear Search Test

String searchTerm = "Book";

Product foundProduct = *linearSearch*(products, searchTerm);

System.*out*.println("Linear Search Result:");

System.*out*.println(foundProduct != null ? foundProduct : "Product not found");

// Sort products by name for Binary Search

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

// Binary Search Test

foundProduct = *binarySearch*(products, searchTerm);

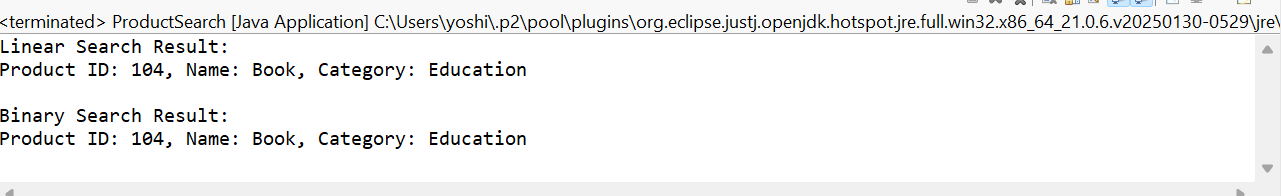
System.*out*.println("\nBinary Search Result:");

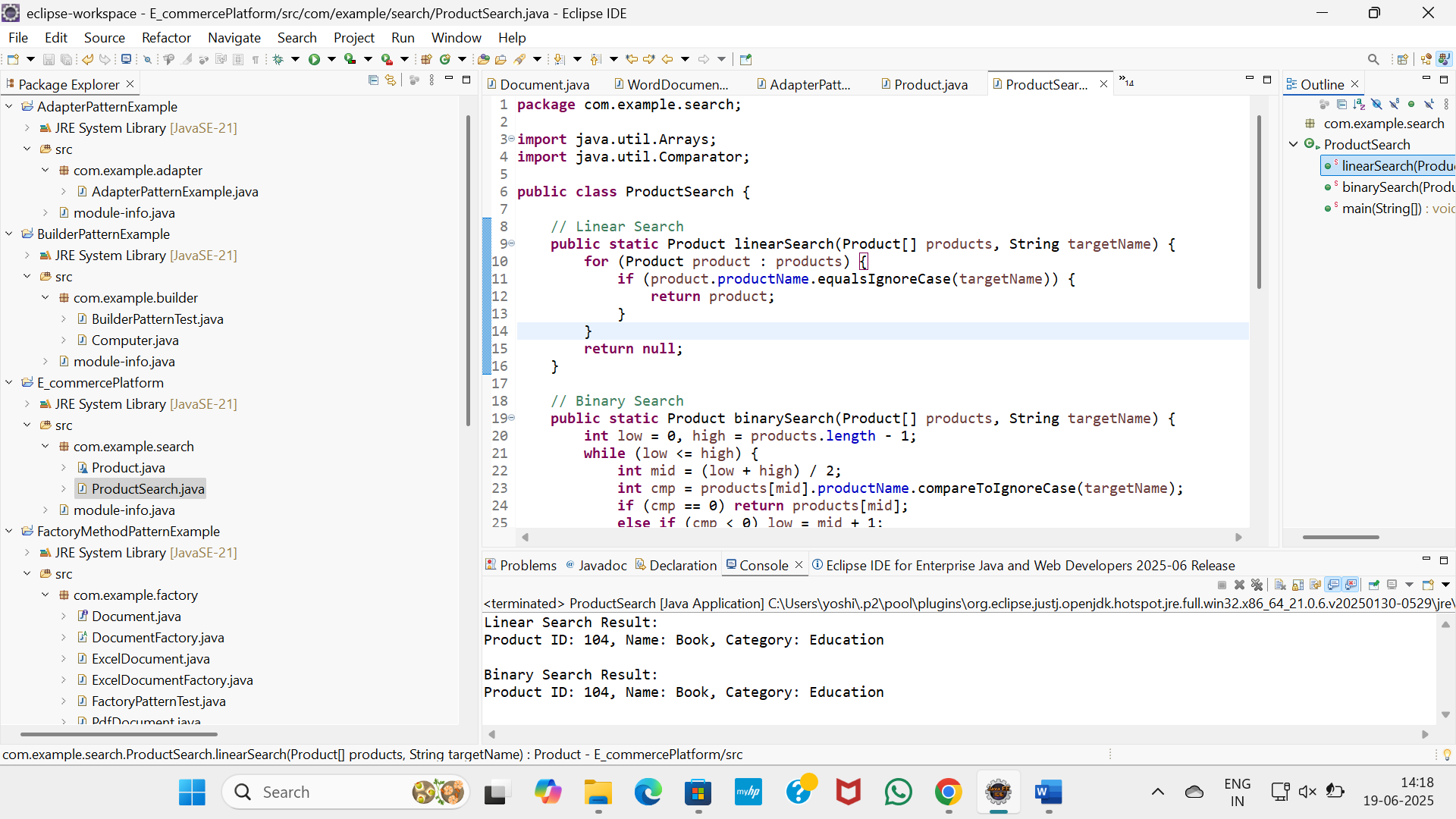
System.*out*.println(foundProduct != null ? foundProduct : "Product not found");

}

}

OUTPUT:



****

**Exercise 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.

1. Understand Recursive Algorithms

Recursion:

* Recursion is a programming technique where a function calls itself to solve smaller instances of a problem.
* It is useful for problems that can be broken down into smaller, similar sub-problems.
* A recursive function must include:
  + A base case – to stop the recursion.
  + A recursive case – where the function calls itself.

Examples:

* Fibonacci numbers
* Factorial computation
* Tree/graph traversal
* Investment growth over time (as in this case)

2. Setup: Recursive Formula

We want to calculate the future value of an investment using the formula:

futureValue = currentValue × (1 + growthRate)^years

We can represent this recursively as:

FV(currentValue, growthRate, years) = (1 + growthRate) × FV(currentValue, growthRate, years - 1)

Base Case: FV(currentValue, growthRate, 0) = currentValue

3. Implementation

Recursive Forecasting Method in Java:

public class FinancialForecast {

// Recursive method

public static double predictFutureValueRecursive(double currentValue, double growthRate, int years) {

if (years == 0) {

return currentValue;

}

return (1 + growthRate) \* predictFutureValueRecursive(currentValue, growthRate, years - 1);

}

// Iterative method (for optimization)

public static double predictFutureValueIterative(double currentValue, double growthRate, int years) {

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

currentValue \*= (1 + growthRate);

}

return currentValue;

}

// Direct calculation using Math.pow

public static double predictUsingMathPow(double currentValue, double growthRate, int years) {

return currentValue \* Math.pow(1 + growthRate, years);

}

public static void main(String[] args) {

double currentValue = 1000.0;

double growthRate = 0.05; // 5% growth per year

int years = 5;

System.out.printf("Recursive Method: ₹%.2f\n", predictFutureValueRecursive(currentValue, growthRate, years));

System.out.printf("Iterative Method: ₹%.2f\n", predictFutureValueIterative(currentValue, growthRate, years));

System.out.printf("Math.pow Method : ₹%.2f\n", predictUsingMathPow(currentValue, growthRate, years));

}

}

4. Analysis

Time Complexity

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Time Complexity | Space Complexity | Remarks |
| Recursive | O(n) | O(n) (call stack) | Simple but can overflow on large n |
| Iterative | O(n) | O(1) | More efficient for large values |
| Math.pow | O(1) | O(1) | Fastest, built-in function |

When to Use Recursion

* When the problem is naturally recursive or involves repeated patterns.
* When simplicity and readability matter more than raw performance.

Optimization Tips

* For large n, avoid recursion due to stack overflow risk.
* Prefer iterative or Math.pow-based solutions in real applications.
* For complex forecasting (e.g., changing rates, inflation, multiple factors), consider dynamic programming or data-driven models.

Conclusion

* Recursive algorithms are conceptually elegant and suitable for modeling repeated processes like compound growth.
* However, for large-scale or production-grade financial forecasting tools, recursive methods should be optimized or replaced with more efficient alternatives.

CODE

package com.example.forecast;

public class FinancialForecast {

// Recursive method to predict future value

public static double predictFutureValueRecursive(double currentValue, double growthRate, int years) {

if (years == 0) {

return currentValue;

}

return (1 + growthRate) \* *predictFutureValueRecursive*(currentValue, growthRate, years - 1);

}

// Iterative method (optimized)

public static double predictFutureValueIterative(double currentValue, double growthRate, int years) {

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

currentValue \*= (1 + growthRate);

}

return currentValue;

}

// using Math.pow

public static double predictUsingMathPow(double currentValue, double growthRate, int years) {

return currentValue \* Math.*pow*(1 + growthRate, years);

}

public static void main(String[] args) {

double currentValue = 1000.0;

double growthRate = 0.05;

int years = 5;

double resultRecursive = *predictFutureValueRecursive*(currentValue, growthRate, years);

double resultIterative = *predictFutureValueIterative*(currentValue, growthRate, years);

double resultMathPow = *predictUsingMathPow*(currentValue, growthRate, years);

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

System.*out*.printf("Future Value (Iterative): ₹%.2f\n", resultIterative);

System.*out*.printf("Future Value (Math.pow): ₹%.2f\n", resultMathPow);

}

}

OUTPUT

