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

Code –

***ECommercePlatformSearch/src/com/ecommerce/functionalities/Product.java***

package com.ecommerce.functionalities;

public class Product {

private int productId;

private String productName;

private String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

@Override

public String toString() {

return "Product [productId=" + productId + ", productName=" + productName + ", category=" + category + "]";

}

}

***ECommercePlatformSearch/src/com/ecommerce/functionalities/LinearSearch.java***

package com.ecommerce.functionalities;

public class LinearSearch {

// Linear Search using productId

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

for (Product product : products) {

if (product.getProductId() == productId) {

return product;

}

}

return null;

}

}

***ECommercePlatformSearch/src/com/ecommerce/functionalities/BinarySearch.java***

package com.ecommerce.functionalities;

public class BinarySearch {

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

return products[mid];

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

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

}

***ECommercePlatformSearch/src/com/ecommerce/functionalities/Main.java***

package com.ecommerce.functionalities;

import java.util.Arrays;

public class Main {

public static void main(String[] args) {

Product[] products = {

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

new Product(105, "Shoes", "Footwear"),

new Product(102, "Phone", "Electronics"),

new Product(107, "Watch", "Accessories"),

new Product(109, "Backpack", "Bags"),

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

new Product(108, "T-shirt", "Clothing")

};

Product[] sortedProducts = Arrays.copyOf(products, products.length);

Arrays.sort(sortedProducts, (p1, p2) -> Integer.compare(p1.getProductId(), p2.getProductId()));

Product found1 = LinearSearch.linearSearch(products, 103);

Product found2 = LinearSearch.linearSearch(products, 104);

Product found3 = BinarySearch.binarySearch(sortedProducts, 107);

Product found4 = BinarySearch.binarySearch(sortedProducts, 106);

System.out.println(found1 != null ? "Linear: Found " + found1.toString() : "Linear: Not Found");

System.out.println(found2 != null ? "Linear: Found " + found2.toString() : "Linear: Not Found");

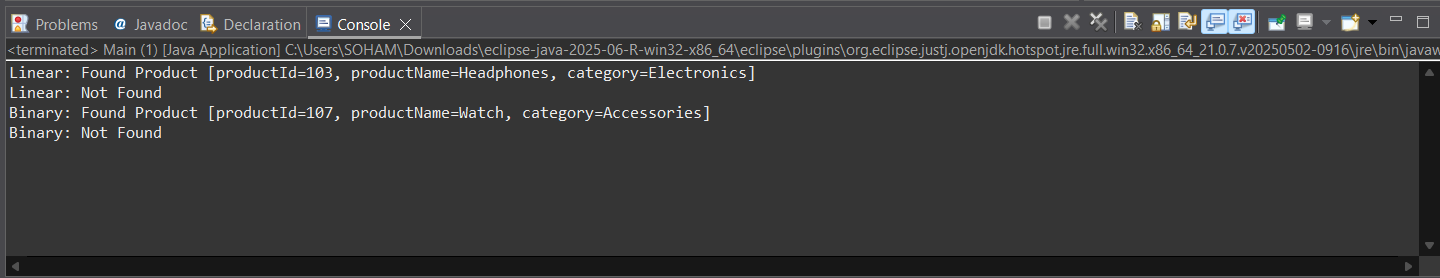
System.out.println(found3 != null ? "Binary: Found " + found3.toString() : "Binary: Not Found");

System.out.println(found4 != null ? "Binary: Found " + found4.toString() : "Binary: Not Found");

}

}

Output –



Answer –

**1. Understand Asymptotic Notation**

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

Big O notation is a mathematical concept used to express the upper limit of an algorithm's time or space complexity in relation to the input size n. It provides a way to evaluate how an algorithm scales as the size of the input increases, emphasizing its efficiency and performance. By ignoring constants and less significant terms, Big O highlights the dominant factor that affects the algorithm's growth rate.

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| **Linear Search** | O(1)- target at the start | O(n)- target in the middle | O(n)- target at the end or not found |
| **Binary Search** | O(1)- target is the mid element | O(log n)- divide array repeatedly | O(log n)- target not found after full division |

**4. Analysis**

*Compare the time complexity of linear and binary search algorithms.*

|  |  |  |  |
| --- | --- | --- | --- |
| **Search Algorithm** | **Time Complexity** | **Space Complexity** | **Requirements** |
| **Linear Search** | O(n) | O(1) | Works on unsorted arrays |
| **Binary Search** | O(log n) | O(log n) (recursive) or O(1) (iterative) | Requires sorted arrays |

*Discuss which algorithm is more suitable for your platform and why.*

Binary Search is more suitable for an e-commerce platform due to the following reasons:

* Inventories are typically large, and performance matters at scale.
* The product list can be sorted once (e.g., by productId) to enable fast searching.
* Binary search offers O(log n) performance, significantly better than O(n) for large datasets.
* Real-time performance is crucial in user-facing search features.

However, Linear Search can still be used for:

* Small data sets like recently viewed products.
* Situations where the data isn't sorted and quick implementation is needed.

**Exercise 7: Financial Forecasting**

Code –

***FinancialForecasting/src/com/FinancialForecastingTool/FinancialTool.java***

package com.FinancialForecastingTool;

public class FinancialTool {

private double futureValue;

public void financialForecast(double presentValue, double growthRate, int timePeriod) {

ForecastingTool future = new ForecastingTool();

futureValue = future.futureForecast(presentValue, growthRate, timePeriod);

System.out.println("Principal Amount="+String.format("%.2f", presentValue)+", Time Period="+timePeriod+","

+ " Annual Growth Rate="+growthRate+", Future Value="+String.format("%.2f", futureValue)+"\n");

}

}

***FinancialForecasting/src/com/FinancialForecastingTool/ForecastingTool.java***

package com.FinancialForecastingTool;

public class ForecastingTool {

public double futureForecast(double presentValue, double growthRate, int timePeriod) {

if (timePeriod <= 0) {

return presentValue;

}

return futureForecast(presentValue, growthRate, (timePeriod - 1)) \* (1 + growthRate);

}

}

***FinancialForecasting/src/com/FinancialForecastingTool/Main.java***

package com.FinancialForecastingTool;

public class Main {

public static void main(String[] args) {

FinancialTool ft = new FinancialTool();

ft.financialForecast(15000.00, 0.055, 10);

ft.financialForecast(85000.50, 0.045, 7);

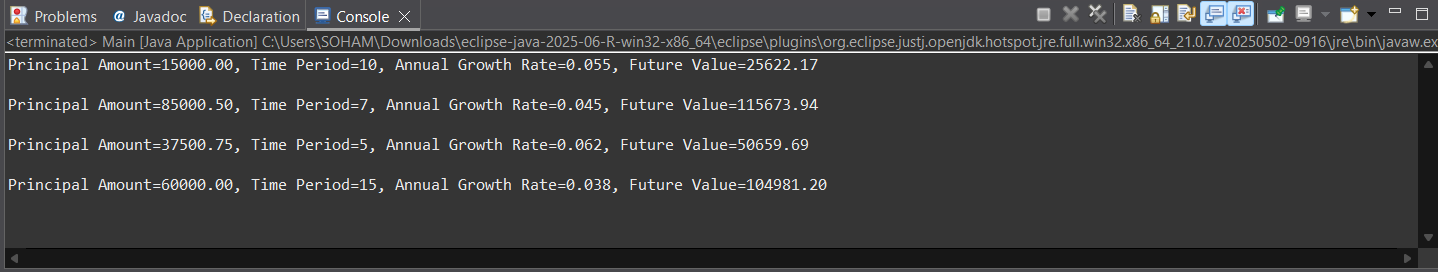
ft.financialForecast(37500.75, 0.062, 5);

ft.financialForecast(60000.00, 0.038, 15);

}

}

Output –



Answer –

**1. Understanding Recursive Algorithms**  
Recursion is a programming technique where a function calls itself to solve smaller sub problems of the original task. Each recursive call breaks the problem down further until it reaches a base case, which stops the recursion.

In financial forecasting, recursion is useful for modeling repetitive calculations, such as computing compound interest. By applying the same formula repeatedly through self-calls, recursive solutions provide a clean and intuitive way to express problems that have a naturally repetitive structure.

1. **Analysis**  
   **Time and Space Complexity**  
   The futureForecast recursive method calculates the future value by making one recursive call for each year. Therefore:

**Time Complexity**: O(n), where n is the number of years (timePeriod)

**Space Complexity**: O(n), due to the recursive call stack growing with each call

Each call performs a constant operation and then recurses with a decremented time period, continuing until it reaches 0.

**Optimizing the Recursive Solution**  
Although the recursive solution is clear and works well for small to medium time periods, it can be improved by converting it into an iterative version. This eliminates the overhead of deep recursion and avoids the risk of stack overflow.

The iterative method uses a loop to multiply the present value year by year, which reduces the **space complexity to O(1)**. In real-world forecasting systems, iterative solutions are generally preferred due to their efficiency and reliability, particularly for long-term projections.