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

Code –

* *ECommercePlatformSearch/src/com/example/search/Product.java*

package com.example.search;

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/example/search/SearchAlgo.java*

package com.example.search;

public interface SearchAlgo {

public Product search(Product[] products, int key);

}

* *ECommercePlatformSearch/src/com/example/search/LinearSearch.java*

package com.example.search;

public class LinearSearch implements SearchAlgo {

@Override

public Product search(Product[] products, int key) {

for (Product product : products) {

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

return product;

}

}

return null;

}

}

* *ECommercePlatformSearch/src/com/example/search/BinarySearch.java*

package com.example.search;

public class BinarySearch implements SearchAlgo {

@Override

public Product search(Product[] products, int key) {

return search(products, key, 0, (products.length - 1));

}

public Product search(Product[] products, int key, int low, int high) {

if (high >= low) {

int mid = (low+high)/2;

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

return products[mid];

}

else if (products[mid].getProductId() > key) {

return search(products, key, low, (mid-1));

}

else {

return search(products, key, (mid+1), high);

}

}

return null;

}

}

* *ECommercePlatformSearch/src/com/example/search/SortedInventory.java*

package com.example.search;

import java.util.Arrays;

public class SortedInventory {

public Product[] productArraySorted(Product[] products) {

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

return products;

}

}

* *ECommercePlatformSearch/src/com/example/search/Test.java*

package com.example.search;

public class Test {

public static void main(String[] args) {

Product[] products = {

new Product(100, "Keyboard", "Electronics"),

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

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

new Product(107, "Pepsi", "Beverages"),

new Product(105, "Ball", "Sports"),

new Product(101, "Pen", "Accessories")

};

Product[] sortedProducts = new SortedInventory().productArraySorted(products);

for (Product product : sortedProducts) {

System.*out*.println(product.toString());

}

System.*out*.println();

SearchAlgo linear = new LinearSearch();

SearchAlgo binary = new BinarySearch();

System.*out*.print("Product with ID 105 (Linear Search): ");

Product p1 = linear.search(products, 105);

System.*out*.println(p1!=null ? p1.toString() : "Not found");

System.*out*.print("Product with ID 103 (Linear Search): ");

Product p4 = linear.search(products, 103);

System.*out*.println(p4!=null ? p4.toString() : "Not found");

System.*out*.print("Product with ID 106 (Binary Search): ");

Product p2 = binary.search(sortedProducts, 106);

System.*out*.println(p2!=null ? p2.toString() : "Not found");

System.*out*.print("Product with ID 101 (Binary Search): ");

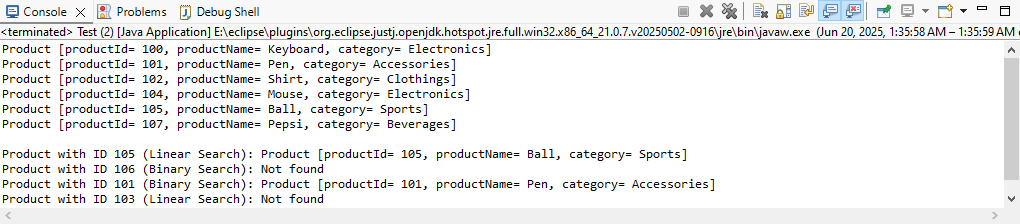
Product p3 = binary.search(sortedProducts, 101);

System.*out*.println(p3!=null ? p3.toString() : "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 tool used to describe the upper bound of an algorithm’s time or space complexity relative to the input size n. It helps developers understand how an algorithm performs as the dataset grows, focusing on the scalability and efficiency of the algorithm. It ignores constants and lower-order terms to capture the 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 –

* *FinancialForecastingTool/src/com/example/recursion/FinancialForecaster.java*

package com.example.recursion;

/\*

\* Assuming formula futureForecast = presentValue \* (1 + growthRate) ^ timePeriod

\*

\* We assume Compound growth over years at a steady rate based on past data

\*/

public class FinancialForecaster {

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

if (timePeriod <= 0) {

return presentValue;

}

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

}

}

* *FinancialForecastingTool/src/com/example/recursion/Forecaster.java*

package com.example.recursion;

public class Forecaster {

private double futureValue;

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

FinancialForecaster future = new FinancialForecaster();

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

System.*out*.println("After " + timePeriod + " years, " +

String.*format*("%.2f", presentValue) + " rupees at " +

String.*format*("%.2f", (growthRate \* 100)) + "% annual growth rate would become " +

String.*format*("%.2f", futureValue) + " rupees");

System.*out*.println();

}

}

}

* *FinancialForecastingTool/src/com/example/recursion/Main.java*

package com.example.recursion;

public class Main {

public static void main(String[] args) {

Forecaster fin = new Forecaster();

fin.forecast(25482.46, 0.0695, 13);

fin.forecast(40000, 0.07, 4);

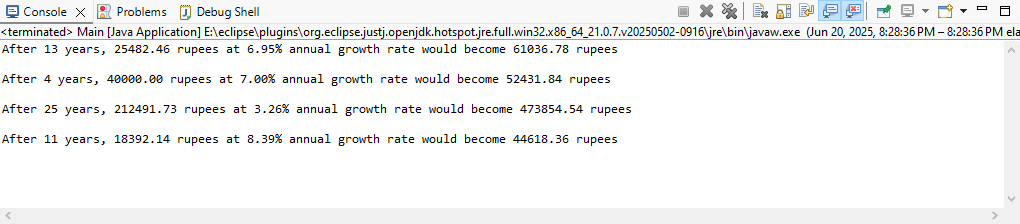
fin.forecast(212491.73, 0.0326, 25);

fin.forecast(18392.14, 0.0839, 11);

}

}

Output –



Answer –

**1. Understand Recursive Algorithms**

*Explain the concept of recursion and how it can simplify certain problems.*

Recursion is a programming technique in which a method calls itself to solve a smaller instance of the same problem. Each recursive call simplifies the problem, bringing it closer to a base case, a condition that stops the recursion.

In financial forecasting, recursion can be used to model repetitive growth calculations, such as compound interest, by breaking down the future value computation into repeated steps based on the same formula. This makes the code concise and easier to understand for problems with natural repetitive structure.

**4. Analysis**

*Discuss the time complexity of your recursive algorithm.*

The recursive futureForecast method computes the future value by making one recursive call for each year (time period). Hence, the time complexity is:

* **Time Complexity**: O(n), where n is the number of years (timePeriod)
* **Space Complexity**: O(n), due to the recursive call stack growing linearly with each call

Each recursive call performs a constant amount of work and then calls itself with a smaller timePeriod, decrementing by 1 each time until reaching 0.

*Explain how to optimize the recursive solution to avoid excessive computation.*

While the current recursion is straightforward and efficient for small to moderate values of timePeriod, it can be optimized using an iterative approach to avoid the overhead of deep recursion and stack usage.

In iterative approach the recursion is replaced with a loop that multiplies the present value step by step over the number of years. This reduces space complexity to O(1). In most practical forecasting tools, an iterative approach is preferred for its better performance and stack safety, especially when forecasting over long time periods.