**WEEK - 2**

**Module 2 : Data Structures and Algorithms**

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

1. **Understanding Asymptotic Notations**

**Big O Notation:**

Big O denotes the upper bound of the time or space complexity of an algorithm or data structure. It gives the upper bound in case of worst-case scenario. It is used to compare the efficiency of algorithms and data structures.

It helps:

* Predict performance for large inputs
* Identify any bottlenecks in the code
* Guide the right selection of algorithm

**Cases for Search Operation:**

1. Linear search:

|  |  |  |
| --- | --- | --- |
| Case | Description | Time Complexity |
| Best | Target is the first element | O(1) |
| Average | Target is in the middle | O(n) |
| Worst | Target is last or not present | O(n) |

2.Binary Search

|  |  |  |
| --- | --- | --- |
| Case | Description | Time Complexity |
| Best | Target is in the middle | O(1) |
| Average | Target is somewhere in list | O(log n) |
| Worst | Target not present | O(log n) |

1. **Setup:**

**CODE:**

import java.util.Arrays;

import java.util.Comparator;

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;

}

public String Printing() {

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

}

}

class LinearSearch {

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

for (Product p : products) {

if (p.productId == targetId) {

return p;

}

}

return null;

}

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

int left = 0, right = products.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

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

return products[mid];

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

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = {

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

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

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

new Product(102, "Notebook", "Stationery"),

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

};

System.out.println("Linear Search (ID = 105):");

System.out.println(linearSearch(products, 105));

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

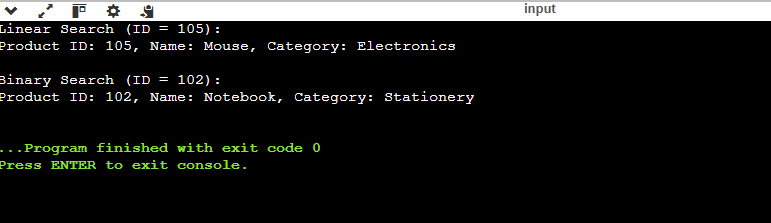
System.out.println("\nBinary Search (ID = 102):");

System.out.println(binarySearch(products, 102));

}

}

**OUTPUT:**



**Exercise 7: Financial Forecasting**

**1.Recursion Concept**

Recursion is a technique where a method calls itself to solve smaller instances of a problem. It is helpful for problems with repeated patterns, such as calculating future investment values over time.

**2. Future Value Forecasting**

public class FinancialForecast {

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

if (years == 0) {

return presentValue;

}

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

}

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

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

presentValue \*= (1 + growthRate);

}

return presentValue;

}

public static void main(String[] args) {

double presentValue = 10000.0;

double growthRate = 0.05;

int years = 10;

double resultRecursive = forecastRecursive(presentValue, growthRate, years);

double resultIterative = forecastIterative(presentValue, growthRate, years);

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

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

**OUTPUT:**  
