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

**Big O Notation:**

* **Big O notation** is used to describe the time or space complexity of an algorithm as the input size (n) grows.
* It helps in understanding how an algorithm performs in terms of execution time or memory usage, regardless of hardware or language.

**Search Operation Scenarios:**

**Best Case:** The target is found immediately (e.g., at the first index).

**Average Case:** The target is found near the middle of the data.

**Worst Case:** The target is at the end or not present, requiring a full array traversal.

**Code:**

import java.util.\*;

public class Product {

    int productId;

    String productName;

    String category;

public Product(int id, String name, String category) {

        this.productId = id;

        this.productName = name;

        this.category = category;

    }

public String toString() {

        return productId + " - " + productName + " (" + category + ")";

    }

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

        for (Product product : products) {

            if (product.productId == targetId) {

                return product;

            }

        }

        return null;

    }

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

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

        while (left <= right) {

            int mid = (left + right) / 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) {

        Scanner sc = new Scanner(System.in);

        Product[] products = {

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

            new Product(205, "Shoes", "Footwear"),

            new Product(150, "Smartphone", "Electronics"),

            new Product(120, "T-shirt", "Apparel"),

            new Product(300, "Watch", "Accessories")

        };

        System.out.print("Enter the Product ID to search: ");

        int targetId = sc.nextInt();

        // Linear Search

        System.out.println("\nLinear Search:");

        Product foundLinear = linearSearch(products, targetId);

        if (foundLinear != null) {

            System.out.println("Product Found: " + foundLinear);

        } else {

            System.out.println("Product Not Found");

        }

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

        // Binary Search

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

        Product foundBinary = binarySearch(products, targetId);

        if (foundBinary != null) {

            System.out.println("Product Found: " + foundBinary);

        } else {

            System.out.println("Product Not Found");

        }

        sc.close();

    }

}

**Output:**

Enter the Product ID to search: 205

Linear Search:

Product Found: 205 - Shoes (Footwear)

Binary Search:

Product Found: 205 - Shoes (Footwear)

**Analysis:**

**Linear search:**

Linear search time complexity is O(n). It is good for small and unsorted arrays.

**Binary search:**

Binary search time complexity is O(log n). It is very efficient for larger and sorted arrays.

“For E-commerce Platform, with a large number of products, Binary Search is preferred for efficiency.”

**Exercise 7: Financial Forecasting**

**Recursion** is a technique where a function calls itself to solve smaller instances of the same problem.

**🔸 Example:**

Instead of using a loop to compute a result, recursion divides the task into smaller sub-tasks.

**🔸 Why Use Recursion?**

* Simplifies code for **divide-and-conquer** problems.
* Useful for mathematical sequences (e.g., Fibonacci, growth rate).
* Elegant but must be used carefully to avoid stack overflow.

**Code:**

import java.util.\*;

public class FinancialForecast {

    public static double calculateFutureValue(double presentValue, double rate, int years) {

        if (years == 0) {

            return presentValue;

        }

        return calculateFutureValue(presentValue \* (1 + rate), rate, years - 1);

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter present value (₹): ");

        double presentValue = sc.nextDouble();

        System.out.print("Enter annual growth rate (%): ");

        double rate = sc.nextDouble() / 100;

        System.out.print("Enter number of years: ");

        int years = sc.nextInt();

        double futureValue = calculateFutureValue(presentValue, rate, years);

        System.out.printf("Future Value after %d years: Rs.%.2f\n", years, futureValue);

        sc.close();

    }

}

**Output:**

Enter present value (?): 15000

Enter annual growth rate (%): 5

Enter number of years: 3

Future Value after 3 years: Rs.17364.38

**Analysis:**

**Time Complexity:**

* The recursion runs once per year, so Time Complexity: O(n) where n is the number of years.

**Space Complexity:**

* Each recursive call adds to the stack → Space Complexity: O(n).

**Optimization:**

**Iterative Approach (if recursion is too deep):**

* Use a loop instead to avoid stack overflow for large n.