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

Big O Notation estimates an algorithm’s efficiency by describing its worst-case performance as input size grows, focusing on key operations regardless of machine or language.

Search Scenarios:

* **Best Case:** Target is found instantly (first position).
* **Average Case:** Target is found around the middle.
* **Worst Case:** Target is last or missing, needing full traversal.

Program:

import java.util.Arrays;

class Product implements Comparable<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 int compareTo(Product other) {

        return this.productName.compareToIgnoreCase(other.productName);

    }

    public String toString() {

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

    }

}

public class EcommerceSearch {

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

        for (Product product : products) {

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

                return product;

            }

        }

        return null;

    }

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

        int left = 0;

        int right = products.length - 1;

        while (left <= right) {

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

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

            if (cmp == 0) {

                return products[mid];

            } else if (cmp < 0) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return null;

    }

    public static void main(String[] args) {

        Product[] products = {

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

            new Product(102, "Shoes", "Footwear"),

            new Product(103, "Watch", "Accessories"),

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

        };

        // Linear Search

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

        Product result1 = linearSearch(products, "watch");

        System.out.println(result1 != null ? result1 : "Product not found");

//Binary Search

        Arrays.sort(products);

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

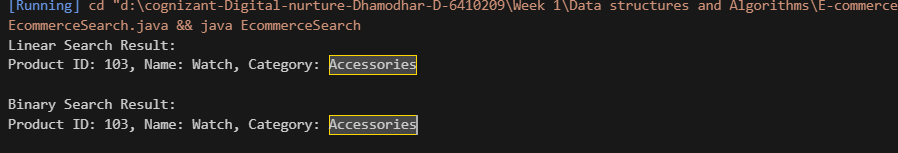
        Product result2 = binarySearch(products, "watch");

        System.out.println(result2 != null ? result2 : "Product not found");

    }

}

Output:



**Linear Search**: O(n) time — suitable for small, unsorted arrays.  
**Binary Search**: O(log n) time — ideal for large, sorted arrays.  
**Conclusion**: For e-commerce platforms with many products, **binary search is more efficient**.

**Exercise 7: Financial Forecasting**

**Understand Recursive Algorithm**

**Recursion** is a programming technique where a method calls itself to solve a problem in smaller chunks. It’s especially useful for tasks like financial forecasting, where future values depend on past outcomes.

Program:

public class FinancialForecast {

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

        if (years == 0) {

            return presentValue;

        }

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

    }

    public static double forecastMemo(double presentValue, double growthRate, int years, double[] memo) {

        if (years == 0) {

            return presentValue;

        }

        if (memo[years] != 0.0) {

            return memo[years];

        }

        memo[years] = forecastMemo(presentValue, growthRate, years - 1, memo) \* (1 + growthRate);

        return memo[years];

    }

    public static void main(String[] args) {

        double initialInvestment = 10000.0;

        double annualRate = 0.07;

        int years = 5;

        System.out.println("Using Basic Recursion:");

        double result1 = forecastValue(initialInvestment, annualRate, years);

        System.out.printf("Projected Value after %d years: %.2f%n", years, result1);

        System.out.println("\nUsing Memoization:");

        double[] memo = new double[years + 1];

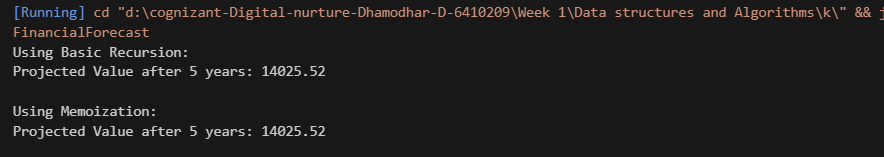
        double result2 = forecastMemo(initialInvestment, annualRate, years, memo);

        System.out.printf("Projected Value after %d years: %.2f%n", years, result2);

    }

}

**Output:**



**Analysis:**

**Time Complexity:**

Basic Recursive Method: O(n), where n is the number of years.

Memoized Version: Still O(n), but avoids redundant calculations.

**Optimization:**

Use **memoization** or convert recursion to an **iterative loop** to reduce stack overhead and improve performance in larger datasets.