**DN 4.0 JAVA FSE SOLUTIONS – WEEK 1**

**SKILL: Data structures and Algorithms**

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

**Step-1: Understanding Asymptotic Notation**

**Big O Notation:**

Big O notation is a way to describe the time or space complexity of an algorithm. It tells us how the performance of an algorithm changes as the input size increases. We don’t worry about exact execution time instead we focus on the growth rate.

Example:

* **O(1)** → Constant time, no matter the size.
* **O(n)** → Linear time, depends directly on input size.
* **O(log n)** → Logarithmic time, very efficient for large inputs.

Big O helps us choose the most efficient algorithm for the task. In real-life projects like an e-commerce platform, efficiency really matters because there could be thousands or millions of products.

**Best, Average, and Worst-Case Scenarios:**

When analyzing search operations:

* **Best Case:** The quickest scenario—like finding the item first for linear search or at the center for binary search.
* **Average Case:** Expected performance for typical inputs across various cases.
* **Worst Case:** The slowest scenario—checking all items for linear search and for binary search, it’s the maximum number of steps needed (log₂n).

**Step-2: Setup**

**Product.java –**

class Product {

    int productId;

    String productName;

    String category;

    double price;

    Product(int productId, String productName, String category, double price) {

        this.productId = productId;

        this.productName = productName;

        this.category = category;

        this.price = price;

    }

    @Override

    public String toString() {

        return productId + " - " + productName + " - " + category + " - $" + price;

    }

}

**Step-3: Implementation**

Implementing linear search and binary search algorithms –

**Search.java** –

import java.util.\*;

public class ECommerceSearch {

   public static Product linearSearchByName(Product[] products, String name) {

        for (Product product : products) {

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

                return product;

            }

        }

        return null;

    }

  public static Product binarySearchByName(Product[] products, String name) {

Arrays.sort(products, Comparator.comparing(p - > p.productName.toLowerCase()));

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

        while (left <= right) {

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

            int result = name.compareToIgnoreCase(products[mid].productName);

            if (result == 0) return products[mid];

            else if (result > 0) left = mid + 1;

            else right = mid - 1;

        }

        return null;

    }

    public static Product linearSearchById(Product[] products, int id) {

        for (Product product : products) {

            if (product.productId == id) {

                return product;

            }

        }

        return null;

    }

    public static Product binarySearchById(Product[] products, int id) {

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

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

        while (left <= right) {

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

            if (products[mid].productId == id) return products[mid];

            else if (id > products[mid].productId) left = mid + 1;

            else right = mid - 1;

        }

        return null;

    }

    public static List<Product> searchByCategory(Product[] products, String category) {

        List<Product> result = new ArrayList<>();

        for (Product product : products) {

            if (product.category.equalsIgnoreCase(category)) {

                result.add(product);

            }

        }

        return result;

    }

}

**TestProduct.java** –

import java.util.\*;

public class TestProduct {

    public static void main(String[] args) {

        Product[] products = {

            new Product(101, "Laptop", "Electronics", 80000.0),

            new Product(102, "Shoes", "Footwear", 2500.0),

            new Product(103, "Watch", "Accessories", 3500.0),

            new Product(104, "Mobile", "Electronics", 60000.0),

            new Product(105, "Sandals", "Footwear", 1200.0)

        };

        System.out.println("Existing Products:");

        for (Product product : products) {

            System.out.println(product);

        }

        Scanner scanner = new Scanner(System.in);

        // Search by Product Name

        System.out.print("\nEnter product name to search: ");

        String searchName = scanner.nextLine();

        Product linearByName = ECommerceSearch.linearSearchByName(products, searchName);

        System.out.println((linearByName != null) ? "Linear Name Search: " + linearByName : "Not found (Linear Name)");

        Product binaryByName = ECommerceSearch.binarySearchByName(products, searchName);

        System.out.println((binaryByName != null) ? "Binary Name Search: " + binaryByName : "Not found (Binary Name)");

        // Search by Product ID

        System.out.print("\nEnter product ID to search: ");

        int searchId = scanner.nextInt();

        Product linearById = ECommerceSearch.linearSearchById(products, searchId);

        System.out.println((linearById != null) ? "Linear ID Search: " + linearById : "Not found (Linear ID)");

        Product binaryById = ECommerceSearch.binarySearchById(products, searchId);

        System.out.println((binaryById != null) ? "Binary ID Search: " + binaryById : "Not found (Binary ID)");

        scanner.nextLine(); // consume newline

        // Search by Category

        System.out.print("\nEnter category to search: ");

        String searchCategory = scanner.nextLine();

        List<Product> categoryResults = ECommerceSearch.searchByCategory(products, searchCategory);

        if (categoryResults.isEmpty()) {

            System.out.println("No products found in category: " + searchCategory);

        } else {

            System.out.println("Products found in category '" + searchCategory + "':");

            for (Product p : categoryResults) {

                System.out.println(p);

            }

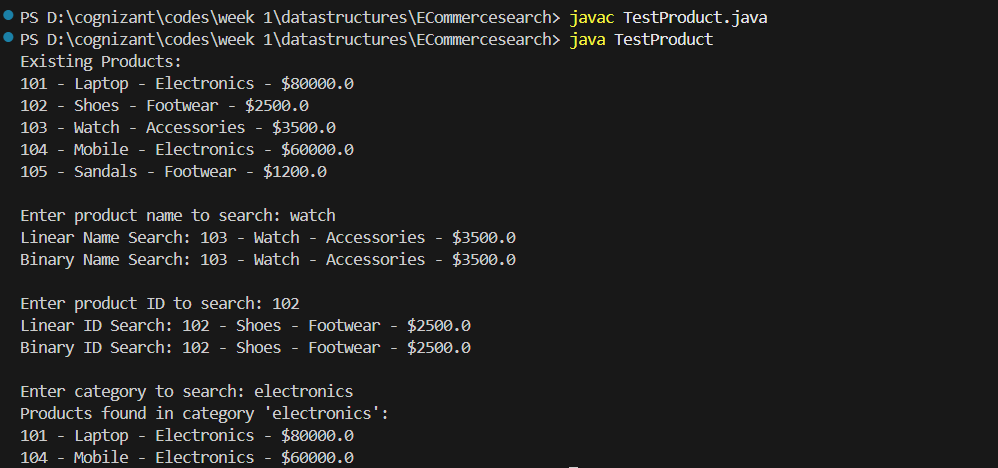
        }

        scanner.close();

    }

}

**Output:**



**Step-4: Analysis**

**Time Complexities:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Search Type** | **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| **Search by Name** | **Linear Search** | **O(1)** | **O(n)** | **O(n)** |
|  | **Binary Search** | **O(1)** | **O(log n)** | **O(log n)** |
| **Search by ID** | **Linear Search** | **O(1)** | **O(n)** | **O(n)** |
|  | **Binary Search** | **O(1)** | **O(log n)** | **O(log n)** |
| **Search by Category** | **Linear Search** | **O(1)** | **O(n)** | **O(n)** |

**Which is Better for E-commerce Search?**

* Binary Search is faster (O(log n)) and best for sorted, rarely changing data like productId or productName. But in this code, sorting is done each time, adding O(n log n) cost.
* Linear Search is slower (O(n)) but works without sorting—ideal for small datasets or frequent updates, especially when searching by category.

**Exercise 2:** Financial Forecasting

**Step 1: Understand Recursive Algorithms**

**Recursion:**

Recursion is a technique in programming where a method calls itself to solve a smaller version of a problem. It's often used when a problem can be broken down into similar sub-problems.

* It makes some problems simpler and more readable.
* Instead of writing loops, you define a base case and a recursive case.

Examples: factorial, Fibonacci, tree traversals, etc.

**Step-2: Setup**

We will forecast the future value of an investment based on:

* Initial value
* Growth rate (percentage)
* Number of years

**Formula:**

FutureValue = InitialValue \* (1 + growthRate)^years

**Step 3: Implementation – Recursive Forecast Function**

**Code:**

import java.util.Scanner;

public class FinancialForecast {

    // Recursive method to calculate future value

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

        if (years == 0) {

            return presentValue; // Base case

        } else {

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

        }

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter initial investment amount (₹): ");

        double initialInvestment = scanner.nextDouble();

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

        double growthRatePercent = scanner.nextDouble();

        double growthRate = growthRatePercent / 100.0; // Convert to decimal

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

        int years = scanner.nextInt();

        // Calculate future value

        double futureValue = calculateFutureValue(initialInvestment, growthRate, years);

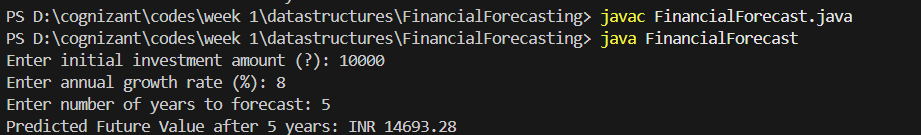
        System.out.printf("Predicted Future Value after %d years: INR %.2f\n", years, futureValue);

        scanner.close();

    }

}

**Output:**

****

**Step 4: Analysis**

**Time Complexity:**

* The recursive function calls itself once per year until years == 0. So, the time complexity is:

O(n), where n = number of years

* It’s linear because we do one operation per recursive call.

**To Optimize the Recursive Solution:**

In this case, recursion is fine because we just do one calculation per year, and the depth of recursion is small (like 5–50 years).

**Iterative Version (Faster for Larger Years):**

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

double result = presentValue;

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

result \*= (1 + growthRate);

}

return result;

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Approach** | **Time Complexity** | **Pros** | **Cons** |
| Recursive | O(n) | Simple and clean logic | Slower for large inputs |
| Iterative | O(n) | More efficient and safer | Slightly more code |

If you use recursion in more complex problems (like Fibonacci), it leads to:

* Repeated computations
* Exponential time, e.g., O(2ⁿ)

**Optimization Techniques:**

* **Memoization**: Store already calculated results (usually for overlapping subproblems)
* **Convert to Iterative**: Use a loop instead of recursion for performance