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

**SKILL: Data structures and Algorithms**

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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Big O Notation:**

A mathematical notation used to describe the performance or complexity of an algorithm, specifically how its runtime or memory usage grows as the input size increases.

It **focuses on the worst-case** growth rate and ignores constants and less significant terms — helping us understand how scalable an algorithm is.

Example O(1),O(n),O(long n).

**Best, average, and worst-case scenarios for search operations**

**Best case** : the function which performs the minimum number of steps on input data of n elements.

**Worst case** : the function which performs the maximum number of steps on input data of size n.

**Average case** : the function which performs an average number of steps on input data of n elements.

**Code:**

**Product.java**

public 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;

    }

    @Override

    public String toString() {

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

    }

}

**Search.java**

import java.util.Arrays;

import java.util.Comparator;

public class Search {

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

         for (int i = 0; i < products.length; i++) {

        if (products[i].productName.equalsIgnoreCase(target))

            return products[i];

         }

       return null;

        }

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

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

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

        while (left <= right) {

            int mid = (left + right) / 2;

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

            if (cmp == 0) {

                return products[mid];

            } else if (cmp < 0) {

                left = mid + 1;

            } else {

                right = mid -1; }   }

       return null;

    }}

**Main.java**

public class Main {

    public static void main(String[] args) {

        Product[] products = {

            new Product(101, "book", "stationary"),

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

            new Product(103, "chappal", "Footware"),

            new Product(104, "Watch", "Accessories"),

            new Product(105, "pencilbox", "stationary")

        };

        String searchword= "chappal”;

        Product linear = Search.linearSearch(products, searchword);

      if (linear != null) {

    System.out.println("Linear Search: " + linear);

} else {

    System.out.println("Linear Search: Not found");

}

        Product binary = Search.binarySearch(products, searchword);

        if (binary!= null) {

    System.out.println("Binary Search: " + binary);

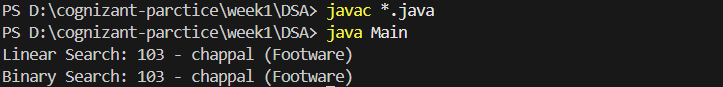
} else {

    System.out.println("Binary Search: Not found");

} }}

**OUTPUT**

For searchword=”chappal



**Analysis**

**time complexity of linear and binary search algorithms.**

|  |  |  |  |
| --- | --- | --- | --- |
| method | Best case | Worst case | Average case |
| Linear search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

**Binary Search is more suitable for E-commerce Search Requirements:**

Because **binary search** is fast and scalable search in an e-commerce platform, especially when the product list is large and sorted (by name, price, or ID).

Use **Linear Search** only in cases where sorting isn't feasible or the dataset is small.

|  |
| --- |

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Solution**

Recursion is a programming concept where a function calls itself within its own definition to solve a problem

3 steps in recursion are

Assumption

Base condition : condition where the function stops calling itself.

Main Logic: The function calls itself with a smaller or simpler input.

**how it can simplify certain problems**

1. Break Problems into Smaller, Repeating Sub-Problems

2. Handle Hierarchical or Nested Structures Easily

3. Great for Backtracking Problems

4. Reduces Code Size and Improves Readability

**Code**

**InvestmentForecast.java**

public class InvestmentForecast {

    public static double calculateFutureAmount(int years, double deposit, double rate) {

        if (years == 0) {

            return 0;

        }

        return (calculateFutureAmount(years - 1, deposit, rate) + deposit) \* (1 + rate);

    }

    public static void main(String[] args) {

        int years = 10;

        double deposit = 50000;

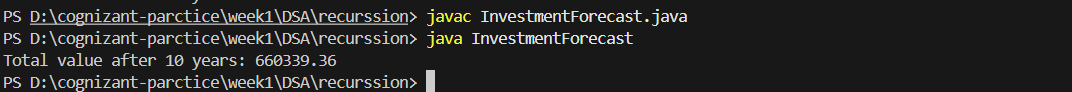
        double rate = 0.05;

        double total = calculateFutureAmount(years, deposit, rate);

        System.out.printf("Total value after %d years: %.2f\n", years, total);

    }}

**OUTPUT:**



**Analysis:**

Time Complexity **: O(n)**

For each year, one recursive call is made.

So the number of calls = years

**Drawback**:

Stack Overflow Risk : If n (number of years) is very large (e.g., 10,000), Java might throw a StackOverflowError due to deep recursion.

**Optimization methods**

1.convert to iteration : Instead of recursion, use a loop to calculate the value — saves memory and prevents stack overflow.

2.memoization : I f there are overlapping subproblems