WEEK 1: Algorithms\_Data Structures

**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.

## **Understanding Asymptotic Notation**

Big O Notation: Big O notation describes how an algorithm’s **runtime or space requirement grows with input size** (n). It helps evaluate efficiency regardless of hardware or specific input.

O(1): Constant time

O(log n): Logarithmic time

O(n): Linear time

O(n log n): Linearithmic

O(n²): Quadratic time

### **Search Operation Scenarios**

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| **Linear Search** | O(1) (first item) | O(n/2) ≈ O(n) | O(n) (last or not found) |
| **Binary Search** | O(1) (middle item) | O(log n) | O(log n) |

**Binary Search requires a sorted array**; linear search works on any unsorted list.

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;

    }

    public String toString() {

        return "[" + productId + "] " + productName + " (" + category + ")";

SearchDemo.java

import java.util.Arrays;

import java.util.Comparator;

public class SearchDemo {

    // Linear search by productId

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

        for (Product product : products) {

            if (product.productId == targetId) return product;

        }

        return null;

    }

    // Binary search by productId (array must be sorted by productId)

    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) {

        Product[] products = {

            new Product(102, "Laptop", "Electronics"),

            new Product(101, "Shoes", "Fashion"),

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

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

        };

        //Linear Search

        System.out.println("Linear Search for ID 104:");

        Product foundLinear = linearSearch(products, 104);

        System.out.println(foundLinear != null ? foundLinear : "Not Found");

        // Sort array for Binary Search

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

        //Binary Search

        System.out.println("\nBinary Search for ID 104:");

        Product foundBinary = binarySearch(products, 104);

        System.out.println(foundBinary != null ? foundBinary : "Not Found");

    }

}

Analysis: Linear vs Binary

| **Algorithm** | **Time Complexity** | **Sorted Array Required** |
| --- | --- | --- |
| Linear Search | O(n) | No |
| Binary Search | O(log n) | Yes |

For small or unsorted product lists, linear search is fine.For large, sorted datasets, binary search is optimal for performance.

**Exercise 7: Financial Forecasting**

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

Understanding Recursive Algorithms

**What is recursion?**

A function is recursive if it calls itself to solve smaller parts of the problem. Each recursive call reduces the problem until it reaches a base case (a condition where recursion stops).

**Why use recursion?**

Simplifies problems that are naturally hierarchical (e.g., tree traversal, factorial, Fibonacci, etc.) Breaks down large problems into smaller, identical sub-problems.

Example of recursion:

Calculating compound growth:

futureValue(years) = futureValue(years - 1) \* (1 + growthRate)

with futureValue(0) = initialValue

**Setup: Recursive Method Structure**

We’ll forecast the future value of an investment recursively:

FV(n) = FV(n - 1) \* (1 + rate)

FinancialForecast.java

public class FinancialForecast {

    // Recursive method to compute future value

    public static double futureValue(double initialValue, double rate, int years) {

        if (years == 0) {

            return initialValue;  // Base case: no growth at year 0

        }

        return futureValue(initialValue, rate, years - 1) \* (1 + rate);

    }

    public static void main(String[] args) {

        double initialValue = 1000;  // Initial investment

        double rate = 0.05;          // 5% growth rate per year

        int years = 10;              // Forecast 10 years into future

        double value = futureValue(initialValue, rate, years);

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

    }

}

**Analysis: Time Complexity**

The time complexity of this recursion:

O(n) → because each call reduces years by 1 and makes one recursive call

No overlapping subproblems, so no excessive recomputation (unlike Fibonacci)

How to Optimize

Although this recursive version is fine (because each call does simple work and no recomputation of overlapping work):

You could optimize by using iteration recursion isn’t necessary for linear problems.

Example iterative:

public static double futureValueIterative(double initialValue, double rate, int years) {

double value = initialValue;

for (int i = 1; i <= years; i++) {

value \*= (1 + rate);

}

return value;

}