Data Structures and Algorithms

# Exercise 02 – E-Commerce platform search function

### Understanding Asymptotic Notations

Big O notation represents the upper bound of the running time of an algorithm. It determines how the runtime or space requirement of an algorithm scales within input size n. It helps in comparing algorithms independently of hardware and understanding scalability and performance of bottlenecks.

If f(n) describes the running time of an algorithm, f(n) is O(g(n)) if there exist a positive constant C and n0 such that, 0 ≤ f(n) ≤ cg(n) for all n ≥ n0

It returns the highest possible output value (big-O) for a given input. The execution time serves as an upper bound on the algorithm's time complexity.

BigO

Best, Average and Worst case in search

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Best Case | Average Case | Worst Case |
| Linear Search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

Best Case: First match.

Average Case: Element is somewhere in the middle.

Worst Case: Not found / at the end (Linear) or max log splits (Binary).

### Setup – Product Class

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 + "]";

    }

}

### Implementation – Linear Search and Binary Search

SearchDemo.java

import java.util.Arrays;

import java.util.Comparator;

public class SearchDemo {

    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 - left) / 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, "Keyboard", "Electronics"),

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

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

            new Product(103, "T-shirt", "Fashion")

        };

        int searchId = 104;

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

        Product result1 = linearSearch(products, searchId);

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

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

        System.out.println("Binary Search (after sorting):");

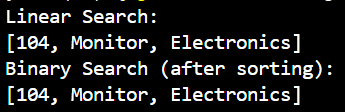
        Product result2 = binarySearch(products, searchId);

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

    }

}

Output:



### Analysis and Comparison

Time Complexity of Linear Search and Binary Search

|  |  |  |
| --- | --- | --- |
| Algorithm | Time Complexity | Space Complexity |
| Linear Search | O(n) | O(1) |
| Binary Search | O(log n) | O(1) |

Binary Search algorithm is more suitable for the E-Commerce platform:

* Performance advantage - Binary search is significantly faster (O (log n)) compared to linear search (O(n)). For 1,000,000 products: Linear search may take up to 1,000,000 comparisons. Binary search will take about log₂ (1,000,000) ≈ 20 comparisons.
* Sorted data is common: E-commerce platforms typically maintain sorted data for fast search, sorting, and filtering (by product ID, price, name, etc.). Product listings are often indexed in databases, which naturally support efficient search trees or binary search structures.
* Better User experience: Fast search = faster results = better user experience. Customers expect real-time or near-instant search results.

# Exercise 07 – Financial Forecasting

### Understanding recursive algorithms

Recursion is a programming technique where a function calls itself to solve a problem.

Each recursive call works on a smaller subproblem, bringing the solution closer with each call until it reaches a base case — a condition that stops further recursion.

### Setup and Implementation

FinancialForecast.java

public class FinancialForecast {

    public static double forecastRecursive(double principal, double rate, int years) {

        if (years == 0) {

            return principal;

        }

        return forecastRecursive(principal, rate, years - 1) \* (1 + rate);

    }

    public static double forecastMemoized(double principal, double rate, int years, Double[] memo) {

        if (years == 0) return principal;

        if (memo[years] != null) return memo[years];

        memo[years] = forecastMemoized(principal, rate, years - 1, memo) \* (1 + rate);

        return memo[years];

    }

    public static void main(String[] args) {

        double principal = 10000;

        double rate = 0.08;

        int years = 5;

        System.out.println("Recursive Forecast:");

        double futureValue = forecastRecursive(principal, rate, years);

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

        System.out.println("\nMemoized Forecast:");

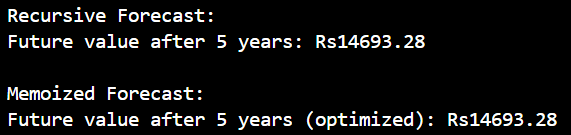
        Double[] memo = new Double[years + 1];

        double futureValueMemo = forecastMemoized(principal, rate, years, memo);

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

    }

}

Output:

### 4. Analysis

Time complexity of recursive algorithm:

|  |  |  |
| --- | --- | --- |
| Method | Time Complexity | Space Complexity |
| Basic Recursion | O(n) | O(n) |
| Memoized Version | O(n) | O(n) |

The basic recursive version makes n recursive calls. Memoized version stores intermediate results, avoiding recomputation.

Optimizing recursive solutions to minimize computation:

* Memoization: Store results of already computed subproblems (as shown above).
* Bottom-Up Approach (DP): Use a loop instead of recursion to build the result iteratively.
* For very large values of n where recursion depth could lead to StackOverflowError.
* In performance-critical real-world financial systems, iterative solutions or precomputed formulas are preferred.