WEEK 1

DATA STRUCTURES AND ALGORITHMS

# Exercise 2: E-commerce Platform Search Function

* Big O notation is a mathematical tool used to describe the efficiency of algorithms by expressing how their time or space requirements grow with input size. In the context of an e-commerce platform, it helps developers analyze and choose the most efficient search algorithms to ensure fast and scalable performance, especially when dealing with large product catalogs**.**
* In search operations, the best case occurs when the product is found immediately, taking O(1) time; the average case involves finding it mid- way, resulting in O(n) for linear search and O(log n) for binary search; the worst case is when the product is last or missing, also yielding O(n) or O(log n). Understanding these helps developers build faster search systems using sorting, indexing, or hashing.

# Code

import java.util.Arrays; import java.util.Comparator; import java.util.Scanner;

class Product { int productId;

String productName; String category;

public Product(int productId, String productName, String category) { this.productId = productId;

this.productName = productName.trim(); this.category = category.trim();

}

@Override

public String toString() {

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

}

}

public class Main {

// Linear Search

public static Product linearSearch(Product[] products, String name) { for (Product p : products) {

if (p.productName.equalsIgnoreCase(name.trim())) { return p;

}

}

return null;

}

// Binary Search (on sorted array)

public static Product binarySearch(Product[] products, String name) { int low = 0, high = products.length - 1;

name = name.trim();

while (low <= high) {

int mid = (low + high) / 2;

Product midProduct = products[mid];

int cmp = midProduct.productName.compareToIgnoreCase(name);

if (cmp == 0) return midProduct; else if (cmp < 0) low = mid + 1; else high = mid - 1;

}

return null;

}

public static void main(String[] args) { Scanner sc = new Scanner(System.in);

System.out.print("Enter number of products: "); int n = sc.nextInt();

sc.nextLine(); // consume newline

Product[] products = new Product[n];

// Input

for (int i = 0; i < n; i++) { System.out.print("Enter Product ID: "); int id = sc.nextInt();

sc.nextLine();

System.out.print("Enter Product Name: "); String name = sc.nextLine();

System.out.print("Enter Category: "); String category = sc.nextLine();

products[i] = new Product(id, name, category);

}

System.out.print("\nEnter product name to search: "); String searchName = sc.nextLine();

// Linear Search

Product resultLinear = linearSearch(products, searchName); if (resultLinear != null) {

System.out.println("Found using Linear Search: " + resultLinear);

} else {

System.out.println("Product not found (Linear Search)");

}

// Sort for Binary Search

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

// Binary Search

Product resultBinary = binarySearch(products, searchName); if (resultBinary != null) {

System.out.println("Found using Binary Search: " + resultBinary);

} else {

System.out.println("Product not found (Binary Search)");

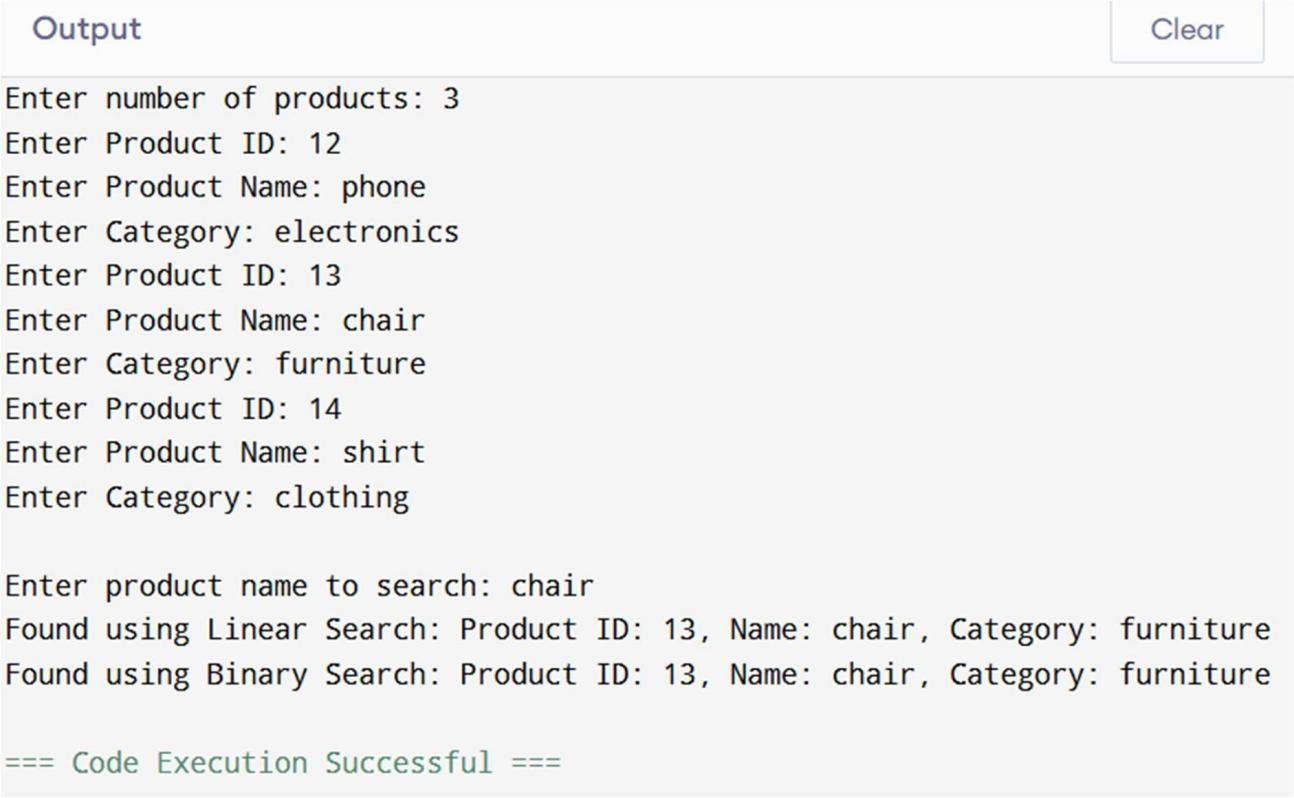
}

sc.close();

}

}

# Output

****

* In the above scenario, the time complexity of linear search is O(1) in the best case (when the product is the first item), but O(n) in both average and worst cases, as it may need to scan through the entire array. On the other hand, binary search offers a much better performance with a best, average, and worst-case time complexity of O(log n), because it repeatedly divides the search space in half. However, binary search requires the product list to be sorted beforehand.
* For an e-commerce platform where the product catalog can be large, binary search is more suitable due to its faster and more scalable performance. Binary search ensures quicker response times for user queries, making it ideal for large-scale applications where performance is critical.

# Exercise 7: Financial Forecasting

* Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller sub-problems. It continues until it reaches a base case, making it useful for tasks like mathematical calculations, tree traversal, and divide-and-conquer algorithms.

# Code

import java.util.\*;

public class Main {

// Naive recursive approach

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

if (years == 0) { return initialValue;

}

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

}

// Optimized recursive approach using memoization

public static double forecastValueMemo(double initialValue, double growthRate, int years, Double[] memo) {

if (years == 0) return initialValue;

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

memo[years] = forecastValueMemo(initialValue, growthRate, years - 1, memo) \* (1 + growthRate);

return memo[years];

}

public static void main(String[] args) { Scanner sc = new Scanner(System.in);

// Taking user input

System.out.print("Enter the initial investment amount: "); double initial = sc.nextDouble();

System.out.print("Enter the annual growth rate (in %): "); double ratePercent = sc.nextDouble();

System.out.print("Enter number of years to forecast: "); int years = sc.nextInt();

// Convert % to decimal

double rate = ratePercent / 100;

// Naive recursive calculation

double futureValue = forecastValue(initial, rate, years);

System.out.println("\n Future Value (recursive): " + String.format("%.2f", futureValue));

// Optimized calculation

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

double futureValueOptimized = forecastValueMemo(initial, rate, years, memo);

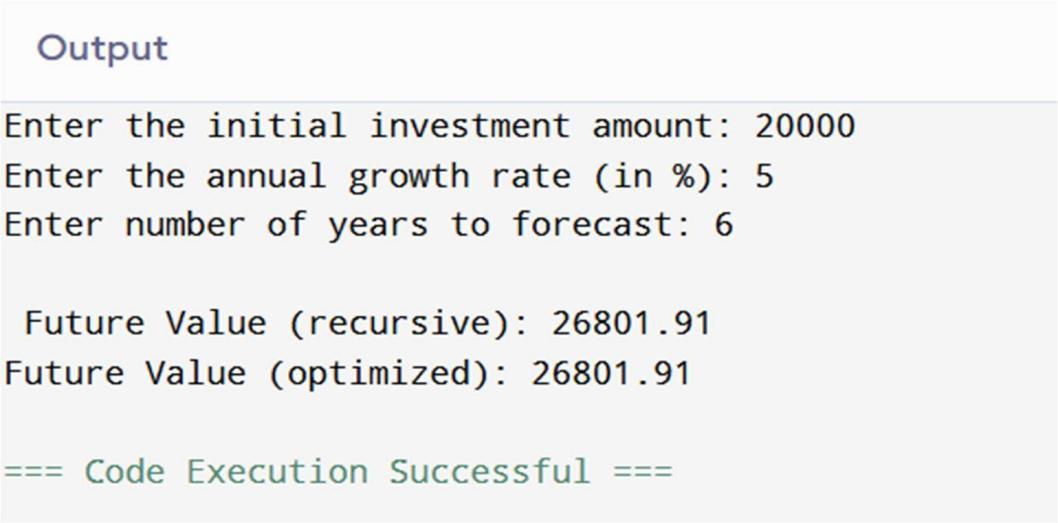
System.out.println("Future Value (optimized): " + String.format("%.2f", futureValueOptimized));

sc.close();

}

}

# Output

****

* The above recursive algorithm calculates the future value of an investment using a naive approach, where the function calls itself once per year until reaching the base case. This results in a time and space complexity of O(n), as each call adds to the call stack and performs a calculation.
* To improve performance, the optimized version uses memoization, which stores previously computed results in an array to avoid redundant calculations. While both approaches have the same theoretical complexity, memoization significantly reduces repeated work, making the algorithm more efficient and better suited for large inputs like long-term investment forecasting.