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

**Solution:**

**Understand Asymptotic Notation:**

What is Big O Notation?

* Big O notation describes the upper bound of an algorithm's runtime as the input size grows.
* It helps us compare and choose the most efficient algorithm.

How it helps in analysing algorithms:

* It helps you analyse and compare algorithms by focusing on their efficiency, especially for large inputs.
* It ignores constants and machine specifics, focusing only on the growth rate.

Describe the best, average, and worst-case scenarios for search operations:

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

**Code:**

import java.util.Arrays;

import java.util.Comparator;

public class Main {

public static void main(String[] args) {

**// Step 3: Store products in an array for linear search**

Product[] products = {

new Product(101, "T-shirt", "Clothing"),

new Product(203, "Laptop", "Electronics"),

new Product(105, "Shoes", "Footwear"),

new Product(150, "Mobile Phone", "Electronics"),

new Product(102, "Watch", "Accessories")

};

// Display input array

System.out.println("Product List (Original Order):");

displayProducts(products);

// Perform Linear Search

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

Product result1 = linearSearch(products, "Shoes");

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

// Step 3: Sort products by productName for binary search

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

// Display sorted array

System.out.println("\n Product List (Sorted by Name for Binary Search):");

displayProducts(products);

// Perform Binary Search

System.out.println("\n Binary Search:");

Product result2 = binarySearch(products, "Shoes");

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

}

// Helper method to display the product array

public static void displayProducts(Product[] products) {

for (Product p : products) {

System.out.println(p);

}

}

**// Linear Search**

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

for (Product p : products) {

if (p.productName.equalsIgnoreCase(targetName)) {

return p;

}

}

return null;

}

**// Binary Search (array must be sorted)**

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

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

while (left <= right) {

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

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

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

if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

}

**// Step 2: Product class with searchable attributes**

class Product {

int productId;

String productName;

String category;

public Product(int id, String name, String category) {

this.productId = id;

this.productName = name;

this.category = category;

}

@Override

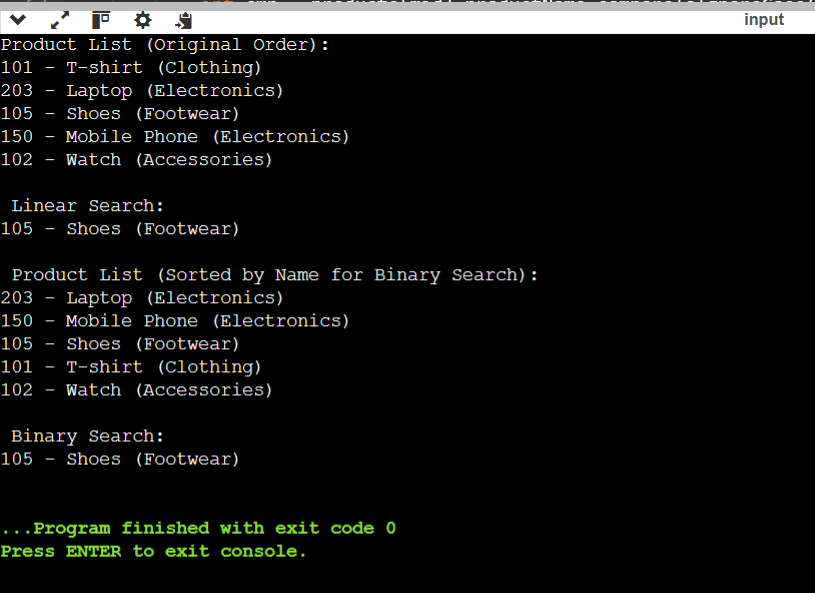
public String toString() {

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

}

}

**Output:**



**Analysis:**

**Comparison of Time Complexity: Linear Search vs Binary Search:**

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

**Which is More Suitable for an E-Commerce Platform?**

Binary Search is more suitable, because:

* E-commerce platforms have large product catalogs, sometimes thousands or millions of items.
* Binary search offers much faster performance (O(log n)) compared to linear search (O(n)) for large datasets.
* Products are usually already sorted or indexed in backend databases (by name, ID, price, etc.), making binary search a practical choice.
* Reduces response time for search queries, improving user experience.

**Exercise 7: Financial Forecasting**

**Solution:**

**Understand Recursion:**

Recursion is a programming technique where a function calls itself to solve a problem by breaking it into smaller subproblems.

**How Recursion Simplifies Problems (Brief Points):**

* Breaks complex problems into simpler, smaller versions.
* Makes code cleaner and easier to understand.
* Naturally fits problems with repetition or hierarchy (e.g., tree traversal).
* Eliminates the need for explicit loops in some cases.
* Useful in mathematical problems like factorial, Fibonacci, and financial forecasting.

**Code:**

public class Main {

public static void main(String[] args) {

double currentValue = 10000;

double growthRate = 0.10;

int years = 5;

// Display input values

System.out.println(" Financial Forecast Input:");

System.out.println("Current Value: ₹" + currentValue);

System.out.println("Annual Growth Rate: " + (growthRate \* 100) + "%");

System.out.println("Forecast Period: " + years + " years\n");

double futureValue = forecastFutureValue(currentValue, growthRate, years);

System.out.printf("Projected future value after %d years: ₹%.2f%n", years, futureValue);

}

**// Step 2: Setup**

**// Create a method to calculate the future value using a recursive approach**

public static double forecastFutureValue(double currentValue, double growthRate, int years) {

**// Step 3: Implementation**

**// Implement a recursive algorithm to predict future values based on past growth rates**

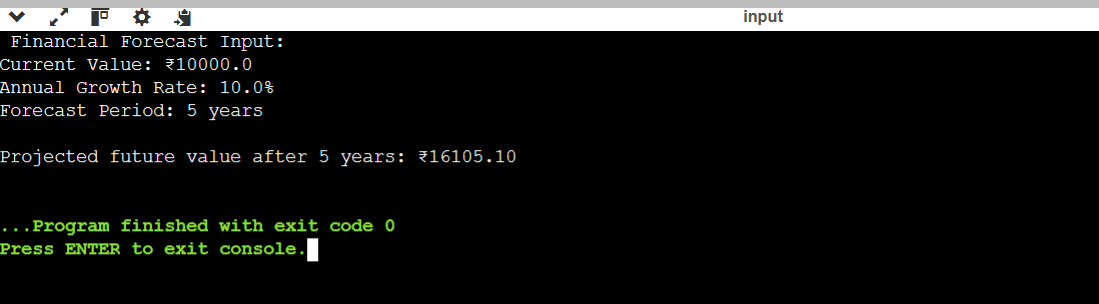
if (years == 0) return currentValue; // base case

return forecastFutureValue(currentValue, growthRate, years - 1) \* (1 + growthRate); // recursive case

}

}

**Output:**



**Analysis:**

**Time Complexity Analysis:**

* The function calls itself once for each year until it reaches 0.
* Therefore, for n years, it performs n recursive calls.

**Time Complexity:**

O(n), where n is the number of years.

**Space Complexity:**

* Due to recursion, each call is stored in the call stack.
* Hence, space complexity is also O(n).

**How to Optimize the Recursive Solution:**

1. **Use Iteration Instead of Recursion**
   * Replace recursion with a simple loop.
   * Saves memory and avoids call stack overflow.
   * Example:

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

currentValue \*= (1 + growthRate);

}

1. **Avoid Memoization Here**
   * Memoization is used to store repeated results.
   * Not needed in this case, since each step is unique.