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

**Understanding Asymptotic Notations :**

Big O notation describes the upper bound of an algorithm's running time or space as a function of input size n. It helps us understand how performance scales with larger inputs.

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

**Binary search** is much faster than linear search but requires the array to be sorted.

**Code :**

import java.util.Arrays;

import java.util.Comparator;

public class SearchFunctionExample {

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

for (Product p : products) {

if (p.productId == targetId) {

return p;

}

}

return null;

}

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(101, "Laptop", "Electronics"),

new Product(205, "Shoes", "Fashion"),

new Product(150, "Book", "Education"),

new Product(300, "Phone", "Electronics"),

new Product(110, "Watch", "Accessories")

};

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

Product result1 = linearSearch(products, 150);

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

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

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

Product result2 = binarySearch(products, 150);

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

}

}

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;

}

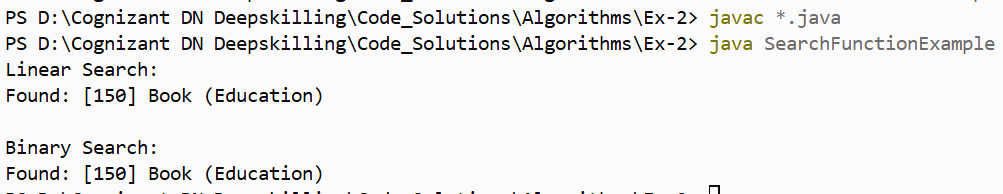
public String toString() {

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

}

}

**Output :**



**Analysis :**

**Linear Search** - Time Complexity: O(n)

**Binary Search** - Time Complexity: O(log n)

For an e-commerce platform, **binary search** is generally **more suitable** because:

* E-commerce platforms usually deal with **large datasets** (thousands of products).
* Data is often **indexed or stored in sorted order** in the backend (e.g., SQL, NoSQL databases).
* **Speed is critical** for user experience — search results must be fast.

**Exercise 7: Financial Forecasting**

**Understanding Recursive Algorithms :**

Recursion is when a method calls itself to solve a smaller instance of the same problem.

It simplifies problems that have a **repetitive or self-similar structure**, like:

* Calculating compound growth
* Fibonacci series
* Tree traversals

In financial forecasting, recursion helps simulate **compound growth** over time.

**Code :**

public class FinancialForecast {

    public static void main(String[] args) {

        double presentValue = 10000;

        double annualGrowthRate = 0.08;

        int years = 5;

        double futureValue = forecastFutureValue(presentValue, annualGrowthRate, years);

        System.out.println("Forecasted Future Value after " + years + " years: Rs. " + futureValue);

    }

    public static double forecastFutureValue(double pv, double rate, int years) {

        if (years == 0) {

            return pv;

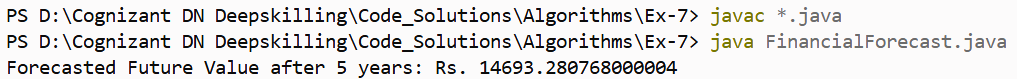
        }

        return forecastFutureValue(pv \* (1 + rate), rate, years - 1);

    }

}

**Output :**



**Analysis :**

Time Complexity = **O(n)**  as,

* Each recursive call reduces years by 1.
* So for n years, we make n recursive calls.

To optimize a recursive financial forecasting solution:

* **Avoid recursion** for large inputs as it can lead to stack overflow.
* **Use iteration** instead, which is more memory-efficient and handles large datasets better.
* **Best option**: Use the mathematical formula FV = PV × (1 + rate)^years with Math.pow() for constant-time performance and no recursion overhead.