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

**1. Understand Asymptotic Notation**

Big O Notation helps describe the upper bound of an algorithm's time or space complexity in terms of input size n. It helps us analyze performance regardless of hardware or language.

* **Linear Search:**
  + Best: O(1) – if the item is at the beginning
  + Average: O(n)
  + Worst: O(n) – if the item is at the end or not present
* **Binary Search:**
  + Best: O(1) – if the item is at the middle
  + Average & Worst: O(log n) – because the array is halved each time
  + Condition**:** The array must be sorted

**2. Setup**

class Product implements Comparable<Product> {

int productId;

String productName;

String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

@Override

public String toString() {

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

}

@Override

public int compareTo(Product other) {

return this.productId - other.productId;

}

}

**3. Implementation**

import java.util.Arrays;

public class ProductSearch {

// Linear Search by Product ID

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

for (Product p : products) {

if (p.productId == targetId) {

return p;

}

}

return null;

}

// Binary Search by Product ID (Array must be sorted)

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(103, "Phone", "Electronics"),

new Product(101, "Laptop", "Electronics"),

new Product(105, "Chair", "Furniture"),

new Product(102, "Book", "Stationery"),

new Product(104, "Table", "Furniture")

};

// Linear search (unsorted array)

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

Product result1 = linearSearch(products, 105);

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

// Binary search (sorted array)

Arrays.sort(products); // Sort by productId

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

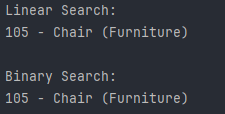
Product result2 = binarySearch(products, 105);

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

}

}

**Output:**



### 4. ****Analysis****

* **Linear Search:**
  + Time Complexity: O(n)
  + Advantage: Works on unsorted data
  + Disadvantage: Slow for large datasets
* **Binary Search:**
  + Time Complexity: O(log n)
  + Advantage: Much faster on large, sorted datasets
  + Disadvantage: Requires sorted input

**Exercise 7: Financial Forecasting**

**1. Understand Recursive Algorithms**

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. It’s especially useful for tasks that can be broken into similar subtasks, such as calculating factorials, Fibonacci numbers, or solving divide-and-conquer problems.

**Advantages of recursion:**

* Simplifies code for problems that have a natural recursive structure.
* Makes code more readable and closer to the mathematical definition.

**2. Setup – Recursive Future Value Calculation**

In financial forecasting, future value (FV) can be calculated using:

**FV = PV × (1 + r)ⁿ**

Where:

* **PV** = Present Value
* **r** = Growth rate (decimal)
* **n** = Number of periods

We can express this formula recursively as: **FV(n) = FV(n - 1) × (1 + r)  
Base case:** When n = 0 → FV = PV

**3. Implementation**

public class FinancialForecast {

// Recursive method to calculate future value

public static double calculateFutureValue(double presentValue, double rate, int years) {

if (years == 0) {

return presentValue;

}

return calculateFutureValue(presentValue, rate, years - 1) \* (1 + rate);

}

public static void main(String[] args) {

double presentValue = 10000; // Initial amount

double annualRate = 0.08; // 8% annual growth

int years = 5; // Forecast for 5 years

double futureValue = calculateFutureValue(presentValue, annualRate, years);

System.out.printf("Future Value after %d years: ₹%.2f", years, futureValue);

}

}

**Output:**

****

**4. Analysis**

**Time Complexity:**  
The recursive function makes one call per year, so the time complexity is **O(n)**, where n is the number of years.

**Optimization Tip:**  
While this recursion is efficient for small n, it can be optimized using:

* **Memoization** (caching previous results).
* **Tail Recursion** (if supported by the compiler).
* **Iterative Approach** for better performance in large-scale simulations.