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

**1. Understanding Asymptotic Notation**

Big O Notation:

Big O notation is a mathematical way to express how an algorithm’s running time or space requirement increases as the size of input data (n) grows. It gives an upper limit on the time taken, helping us measure performance for large datasets.

Examples:

* Linear Search: Time complexity is O(n) – execution time increases linearly with input size.
* Binary Search: Time complexity is O(log n) – much faster for larger, sorted datasets.

**2. Java Class Setup**

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;

}

@Override

public String toString() {

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

}

}

**3. Search Implementations**

Linear Search

public class SearchFunctions {

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

for (Product p : products) {

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

return p;

}

}

return null;

}

}

Binary Search

import java.util.Arrays;

import java.util.Comparator;

public class SearchFunctions {

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

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

while (left <= right) {

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

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

if (cmp == 0)

return products[mid];

else if (cmp < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

public static void sortByName(Product[] products) {

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

}

}

**4. Analysis**

Time Complexity Comparison

|  |  |  |  |
| --- | --- | --- | --- |
| 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) |

E-Commerce Application Suitability:

* Linear Search is easy to implement but inefficient with growing product data.
* Binary Search is much faster but requires a sorted list before every search, which adds maintenance overhead.

Recommendation:  
Use Binary Search for better performance in large catalogs. For real-time and complex search scenarios, implement hash maps for instant lookups or adopt full-text search tools like Elasticsearch for advanced filtering.

**Exercise 7: Financial Forecasting**

**1. Recursion – A Brief Introduction**

Definition:  
Recursion is a method in programming where a function solves a problem by calling itself with smaller inputs, continuing until it reaches a simple, non-recursive base case.

Why Use Recursion?

* Cleaner code for repetitive or nested calculations.
* Ideal for tree structures, compound interest, and other compound growth problems.
* Fits naturally for financial scenarios like year-on-year growth.

For financial forecasting, recursion can be used to repeatedly apply a growth rate over a number of time periods.

**2. Setup: Future Value Formula**

To estimate future value, we use:

                   Future Value = Present Value × (1 + Growth Rate)^n

Where:

* Present Value: Initial amount of money
* Growth Rate: Annual increase rate (in decimal form)
* n: Time period (e.g., years)

**3. Java Code Implementation**

public class FinancialForecast {

public static double futureValue(double current, double rate, int years) {

if (years == 0) {

return current;

}

return futureValue(current \* (1 + rate), rate, years - 1);

}

public static double futureValueOptimized(double current, double rate, int years) {

return current \* Math.pow(1 + rate, years);

}

public static void main(String[] args) {

double currentValue = 10000.0;

double growthRate = 0.07; // 7% annual growth

int years = 5;

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

double usingFormula = futureValueOptimized(currentValue, growthRate, years);

System.out.println("Recursive Result: $" + usingRecursion);

System.out.println("Optimized Result: $" + usingFormula);

}

}

**4. Performance Analysis**

Recursive Version:

* Time: O(n) — one call per year
* Space: O(n) — stack frame per call

Optimized Version:

* Time & Space: O(1) — single calculation using Math.pow()

Optimization Ideas:

* Memoization: Store already computed values to avoid repeating calculations.
* Iterative Method: Replace recursion with a loop to reduce space usage.
* Math-based Solution: The power formula is ideal for clean and efficient forecasts.