**EXERCISE 2: E-COMMERCE PLATFORM SEARCH FUNCTION**

1. **Big O Notation:**

Big O notation is used to describe the efficiency of an algorithm in terms of time or space complexity as the input size (n) grows. It provides an upper bound on the growth rate, that helps us to understand how algorithm works as data changes.

**Common Big O notations:**

* O(1) - constant time
* O(log n) – logarithmic time
* O(n) – linear time
* O(n log n) - log linear time
* O(n^2) – quadratic time

**Search Case Scenarios:**

* **Best Case:** Minimum time it takes (e.g., item is at the start).
* **Average Case:** Average expected time over many searches.
* **Worst Case:** Maximum time taken (e.g., item is at the end or not found)

1. **Setup:**

class Product {

int id;

String name;

String category;

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

this.id = id;

this.name = name;

this.category = category;

}

public String toString() {

return id + ": " + name + " (" + category + ")";

}

}

1. **Implementation:**

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

for (Product p : products) {

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

return p;

}

}

return null;

}

// Binary Search (requires sorted array)

static Product binarySearch(Product[] products, String name) {

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

int compare = products[mid].name.compareToIgnoreCase(name);

if (compare == 0)

return products[mid];

else if (compare < 0)

low = mid + 1;

else

high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

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

new Product(102, "Shoes", "Fashion"),

new Product(103, "Phone", "Electronics"),

new Product(104, "Watch", "Accessories")

};

Product found1 = linearSearch(products, "Phone");

System.out.println("Linear Search Result: " + (found1 != null ? found1 : "Not found"));

Arrays.sort(products, (a, b) -> a.name.compareToIgnoreCase(b.name));

Product found2 = binarySearch(products, "Phone");

System.out.println("Binary Search Result: " + (found2 != null ? found2 : "Not found"));

}

}

**Output:**

Linear Search Result: 103: Phone (Electronics)

Binary Search Result: 103: Phone (Electronics)

1. **Analysis:**

**Time complexity:**

Linear search : O(n)

Binary search : O(log n)

** Linear search** is suitable for **small datasets** or **unsorted data.**

** Binary search** is optimal for **large, sorted datasets** due to its **logarithmic time complexity.**

 For an **e-commerce platform** where product data can grow significantly**, binary search** is **more efficient** if the data is **kept sorted.**

**EXERCISE 7 : FINANCIAL FORECASTING**

1. **Recursive algorithm :**

* Recursion is a method where a function calls itself to solve a problem by breaking it into smaller, simpler subproblems. It's commonly used for tasks like computing factorials, traversing trees, or solving mathematical problems like Fibonacci numbers.
* In financial forecasting, recursion helps model problems where the current state depends on previous ones — like future values depending on past values and growth rates.

1. **Setup:**

**Recursive formula:**

futureValue(years) = futureValue(years - 1) × (1 + growthRate)

**Recursive method:**

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

if (years == 0) {

return initialValue;

} else {

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

}

}

1. **Implementation:**

public class FinancialForecast {

static double forecast(double initialValue, double growthRate, int years) {

if (years == 0) {

return initialValue;

} else {

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

}

}

public static void main(String[] args) {

double initialInvestment = 10000;

double growthRate = 0.10;

int years = 5;

double futureValue = forecast(initialInvestment, growthRate, years);

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

}

}

**Output :**

Future Value after 5 years: 16105.10

1. **Analysis:**

#### Time Complexity:

* **T(n) = T(n-1) + O(1)**
* **Time Complexity: O(n)** → One recursive call

**Optimization:**

public static double forecastIterative(double baseValue, double growthRate, int years) {

double result = baseValue;

for (int i = 1; i <= years; i++) {

result \*= (1 + growthRate);

}

return result;

}

 **Time complexity:** O(n)

 **Space complexity:** O(1)

 **Recursion** provides a natural, elegant way to model financial forecasting based on previous data. But for performance in large datasets, **iteration** is preferred. This technique is ideal for predicting revenue, expenses, or any metric that grows consistently year over year.