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

**2.1 Big O notation:**

Big O notation is a mathematical representation used to describe the time or space complexity of an algorithm in terms of input size n. It helps in analyzing the performance and scalability of algorithms by focusing on their growth rate, not exact timing. Big O allows developers to compare algorithms and choose the most efficient one, especially for large inputs.

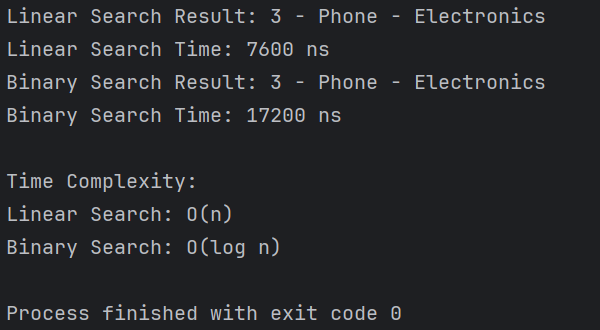
**2.2 The best, average, and worst-case scenarios for search operations:**

* **Linear Search:**
* Best Case: O(1) –when the item is found at the first position.
* Average Case: O(n/2) ≈ O(n) – when the item is somewhere in the middle.
* Worst Case: O(n) – when the item is at the end or not present at all.
* **Binary Search:**
* Best Case: O(1) – when the item is at the middle index.
* Average Case: O(log n) – reduces search space in half each time.
* Worst Case: O(log n) – continues halving until item is found or search space is empty.

**2.3 Code:**

**ECommerceSearch.java**

import java.util.Arrays;  
import java.util.Comparator;  
  
public class ECommerceSearch {  
   
 static 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;  
 }  
 }  
  
 public static Product linearSearch(Product[] products, String name) {  
 for (Product product : products) {  
 if (product.productName.equalsIgnoreCase(name)) {  
 return product;  
 }  
 }  
 return null;  
 }  
  
 public static Product binarySearch(Product[] products, String name) {  
 int left = 0, right = products.length - 1;  
  
 while (left <= right) {  
 int mid = (left + right) / 2;  
 int cmp = name.compareToIgnoreCase(products[mid].productName);  
  
 if (cmp == 0) return products[mid];  
 if (cmp < 0) right = mid - 1;  
 else left = mid + 1;  
 }  
  
 return null;  
 }  
  
 public static void sortByName(Product[] products) {  
 Arrays.*sort*(products, Comparator.*comparing*(p -> p.productName.toLowerCase()));  
 }  
  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(1, "Laptop", "Electronics"),  
 new Product(2, "Shirt", "Clothing"),  
 new Product(3, "Phone", "Electronics"),  
 new Product(4, "Shoes", "Footwear"),  
 new Product(5, "Watch", "Accessories")  
 };  
  
 String searchName = "Phone";  
  
  
 long start1 = System.*nanoTime*();  
 Product result1 = *linearSearch*(products, searchName);  
 long end1 = System.*nanoTime*();  
 System.*out*.println("Linear Search Result: " + (result1 != null ? result1 : "Not Found"));  
 System.*out*.println("Linear Search Time: " + (end1 - start1) + " ns");  
  
  
 *sortByName*(products);  
 long start2 = System.*nanoTime*();  
 Product result2 = *binarySearch*(products, searchName);  
 long end2 = System.*nanoTime*();  
 System.*out*.println("Binary Search Result: " + (result2 != null ? result2 : "Not Found"));  
 System.*out*.println("Binary Search Time: " + (end2 - start2) + " ns");  
  
 System.*out*.println("\nTime Complexity:");  
 System.*out*.println("Linear Search: O(n)");  
 System.*out*.println("Binary Search: O(log n)");  
 }  
}

**2.4 Output:**

**2.5 Analysis:**

Binary search is more suitable for an e-commerce platform because it offers significantly faster performance (O(log n)) compared to linear search (O(n)), especially when dealing with large, sorted product catalogs. Since most platforms maintain structured and indexed data, binary search ensures quicker results and a better user experience.

**Exercise 7: Financial Forecasting**

**7.1 Recursive Algorithm:**

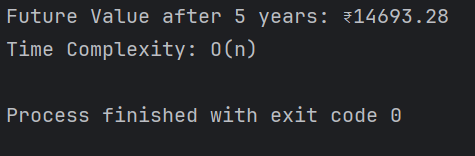
* Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem.
* It simplifies complex problems by breaking them into subproblems that have the same structure.
* For example, in financial forecasting, if we know the current value and the growth rate, we can recursively calculate the future value by applying the growth rate for n periods.

**7.2 Code:**

**FinancialForecast.java**

public class FinancialForecast {  
  
  
 public static double forecast(double currentValue, double rate, int years) {  
 if (years == 0) return currentValue;  
 return *forecast*(currentValue, rate, years - 1) \* (1 + rate);  
 }  
  
  
 public static void main(String[] args) {  
 double currentValue = 10000;  
 double rate = 0.08;  
 int years = 5;  
  
 double futureValue = *forecast*(currentValue, rate, years);  
 System.*out*.printf("Future Value after %d years: ₹%.2f\n", years, futureValue);  
  
  
 System.*out*.println("Time Complexity: O(n)");  
 System.*out*.println("Recursive approach is simple but may be inefficient for large n.");  
  
 }  
}

**7.3 Output:**



**7.4 Analysis:**

The time complexity of the recursive financial forecasting algorithm is O(n), where *n* is the number of periods, because the function makes one recursive call per period. Although this approach is simple and readable, it can lead to excessive stack usage and potential stack overflow for large values of *n* due to its linear recursion depth. To optimize this and avoid excessive computation, we can convert the recursion into an iterative solution, which maintains the same time complexity of O(n) but reduces the space complexity from O(n) to O(1) by eliminating the recursive call stack. This makes the algorithm more efficient and suitable for large-scale forecasting tasks.