Algorithms and Data Structures – Hands-on Exercises

# Exercise 2: E-commerce Platform Search Function

To enhance the user experience on an e-commerce platform, it's important to implement fast and reliable search functionality. This exercise compares linear and binary search techniques to find products efficiently.

Big O Notation expresses the time or space complexity of an algorithm in terms of input size (n). It describes the upper bound, helping analyze scalability and performance.

**Best case**: Minimum time taken (e.g., item found at start).  
**Average case**: Expected time across all inputs.  
**Worst case**: Maximum time taken (e.g., item not found).

## Product.java

public 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;  
 }  
}

## LinearSearch.java

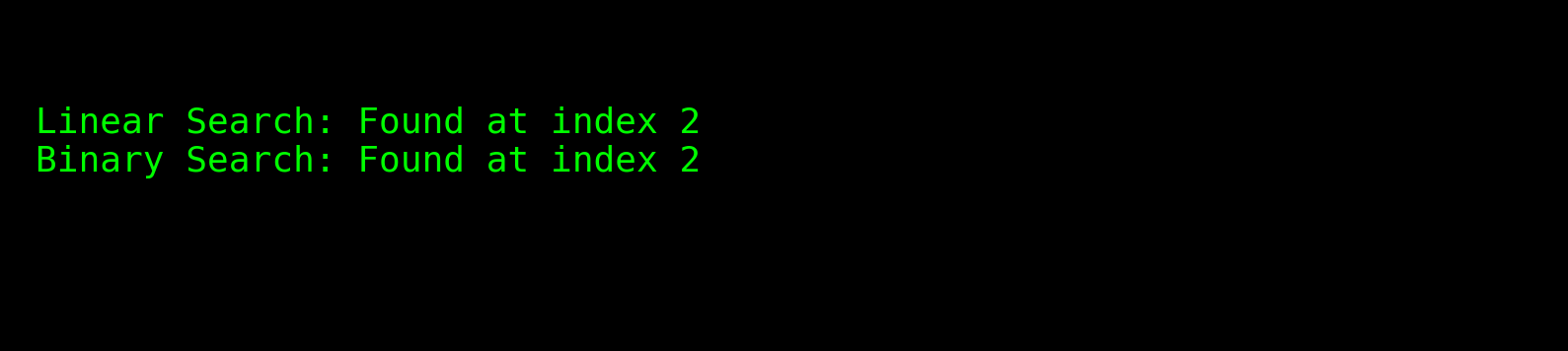
public class LinearSearch {  
 public static int linearSearch(Product[] products, String key) {  
 for (int i = 0; i < products.length; i++) {  
 if (products[i].productName.equalsIgnoreCase(key)) {  
 return i;  
 }  
 }  
 return -1;  
 }  
}

## BinarySearch.java

import java.util.Arrays;  
import java.util.Comparator;  
  
public class BinarySearch {  
 public static int binarySearch(Product[] products, String key) {  
 Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));  
 int left = 0, right = products.length - 1;  
  
 while (left <= right) {  
 int mid = (left + right) / 2;  
 int result = products[mid].productName.compareToIgnoreCase(key);  
  
 if (result == 0) return mid;  
 else if (result < 0) left = mid + 1;  
 else right = mid - 1;  
 }  
 return -1;  
 }  
}

## Main.java

public class Main {  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(101, "Laptop", "Electronics"),  
 new Product(102, "Shoes", "Fashion"),  
 new Product(103, "Book", "Education")  
 };  
  
 int linearResult = LinearSearch.linearSearch(products, "Book");  
 int binaryResult = BinarySearch.binarySearch(products, "Book");  
  
 System.out.println("Linear Search: Found at index " + linearResult);  
 System.out.println("Binary Search: Found at index " + binaryResult);  
 }  
}

→ Screenshot of output :  


**Analysis:**

Linear Search has a time complexity of O(n) — it checks each element one by one.  
BinarySearch has a time complexity of O(log n) — it divides the search space in half each time but requires a sortedarray.  
For unsorted data, Linear Search is better as Binary Search needs sorting first.  
For sorted data or frequent searches, Binary Search is faster and more efficient, making it more suitable for performance-critical platforms.

# Exercise 7: Financial Forecasting – Recursion

This exercise uses recursion to forecast future financial values based on a fixed annual growth rate. It's a simple example showing how recursive methods can simplify computations involving repeated growth.

1: Understand recursive Algorithm:

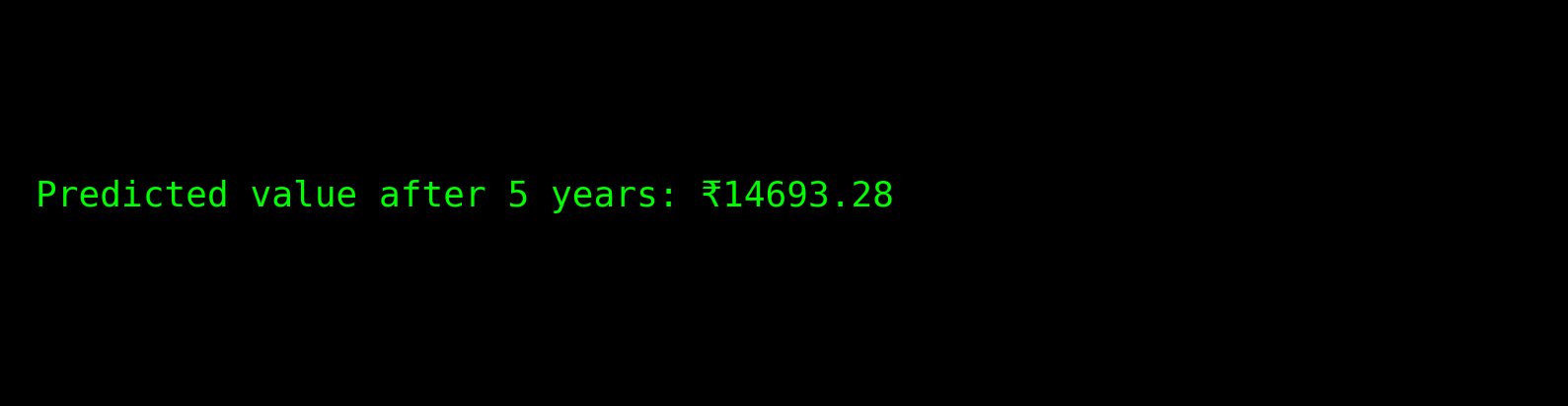
Recursion is a method where a function calls itself to solve smaller instances of a problem. It simplifies problems like tree traversal, factorial, or Fibonacci by breaking them into subproblems, making code cleaner and easier to understand.

## Forecast.java

public class Forecast {  
 public static double predictFutureValue(double presentValue, double rate, int years) {  
 if (years == 0) return presentValue;  
 return predictFutureValue(presentValue \* (1 + rate), rate, years - 1);  
 }  
}

## Main.java

public class Main {  
 public static void main(String[] args) {  
 double presentValue = 10000;  
 double rate = 0.08; // 8% growth rate  
 int years = 5;  
  
 double futureValue = Forecast.predictFutureValue(presentValue, rate, years);  
 System.out.printf("Predicted value after %d years: ₹%.2f\n", years, futureValue);  
 }  
}

→ Screenshot of output :  


Analysis:

Timecomplexity of a recursive algorithm depends on how many times it calls itself and how much work is done per call. For example, naive Fibonacci is O(2ⁿ) due to repeated calls.

To optimize, use:

* Memoization: Store results of subproblems to avoid recomputation.
* Bottom-up DP: Replace recursion with iteration.
* Tail recursion: If supported, allows compiler optimizations.