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

**Product.class:**  
public class Product {

   int productId;

   String productName;

   String category;

   public Product(int var1, String var2, String var3) {

      this.productId = var1;

      this.productName = var2;

      this.category = var3;

   }

   public String toString() {

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

   }

}

**SearchEngine.class:**  
public class SearchEngine {

   public SearchEngine() {

   }

   public static Product linearSearch(Product[] var0, String var1) {

      Product[] var2 = var0;

      int var3 = var0.length;

      for(int var4 = 0; var4 < var3; ++var4) {

         Product var5 = var2[var4];

         if (var5.productName.equalsIgnoreCase(var1)) {

            return var5;

         }

      }

      return null;

   }

   public static Product binarySearch(Product[] var0, String var1) {

      int var2 = 0;

      int var3 = var0.length - 1;

      while(var2 <= var3) {

         int var4 = var2 + (var3 - var2) / 2;

         int var5 = var0[var4].productName.compareToIgnoreCase(var1);

         if (var5 == 0) {

            return var0[var4];

         }

         if (var5 < 0) {

            var2 = var4 + 1;

         } else {

            var3 = var4 - 1;

         }

      }

      return null;

   }

}

**EcommerceSearchDemo.class:**  
import java.util.Arrays;

import java.util.Comparator;

public class EcommerceSearchDemo {

   public EcommerceSearchDemo() {

   }

   public static void main(String[] var0) {

      Product[] var1 = new Product[]{new Product(101, "Laptop", "Electronics"), new Product(102, "Shoes", "Fashion"), new Product(103, "Phone", "Electronics"), new Product(104, "Bag", "Accessories"), new Product(105, "Watch", "Fashion")};

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

      Product var2 = SearchEngine.linearSearch(var1, "Phone");

      System.out.println(var2 != null ? var2 : "Product not found");

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

      Arrays.sort(var1, Comparator.comparing((var0x) -> {

         return var0x.productName.toLowerCase();

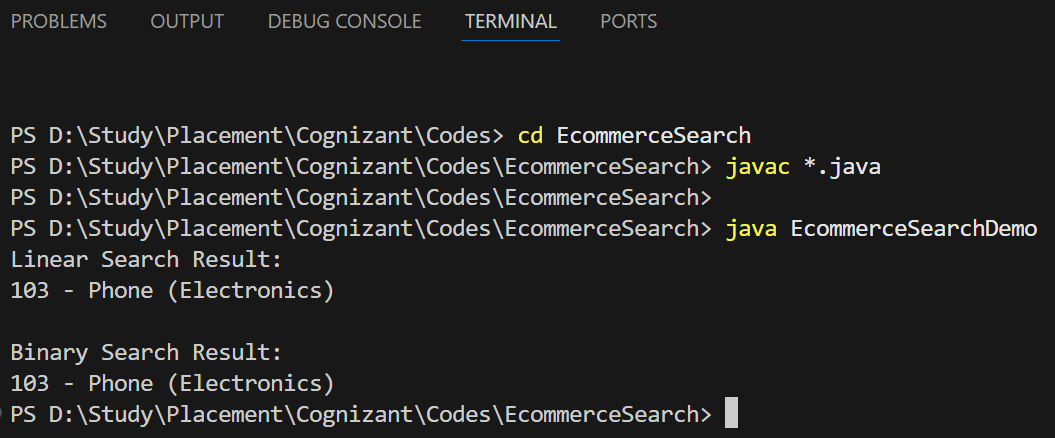
      }));

      Product var3 = SearchEngine.binarySearch(var1, "Phone");

      System.out.println(var3 != null ? var3 : "Product not found");

   }

}

**Output:**  


**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) |

Linear Search: Time grows linearly as list grows. Simple but slow on large lists.

Binary Search: Logarithmic performance. MUCH faster for big sorted datasets.

If product data is **already sorted or can be sorted during preprocessing, binary search** is **way more efficient.**

But if data is **constantly changing**, or small in size, and sorting every time is not worth the effort, **linear search** is fine.

**Exercise 7: Financial Forecasting**

#### **1. What is Recursion?**

**Recursion** is a technique in programming where a method calls itself to solve smaller instances of a problem. It’s like breaking a big problem into smaller bite-sized versions of the same thing, kind of like peeling an onion layer by layer.

**A recursive function has two main parts:**

1. **Base Case:** The stopping point where the recursion ends.
2. **Recursive Case:** The part where the function keeps calling itself.

**We use recursion to:**

1. To simplify code that involves repetitive patterns.

2. Useful in problems like factorial, Fibonacci series, and in our case — **financial forecasting** based on repeated percentage growth.

**2. Setup:**  
Consider, we have a present amount of money (let’s say ₹10,000) and want to know how much it’ll grow after a few years, assuming a constant growth rate (e.g., 8% annually).

We use the formula:

**Future Value (FV) = PV × (1 + r)^n**

Where:

* **FV** = future value
* **PV** = present value
* **r** = growth rate per period (e.g., 0.08 for 8%)
* **n** = number of periods (e.g., years)

This formula has a repetitive pattern, it multiplies the value again and again by the growth rate. This makes it a perfect fit for recursion.

**3. Implementation:**

**FinancialForecast.java:**public class FinancialForecast

{

    public static double calculateFutureValue(double presentValue, double growthRate, int periods)

    {

        if (periods==0)

            return presentValue;

        return calculateFutureValue(presentValue, growthRate, periods - 1) \* (1 + growthRate);

    }

    public static void main(String[] args)

    {

        double presentValue=10000;

        double annualGrowthRate=0.08;

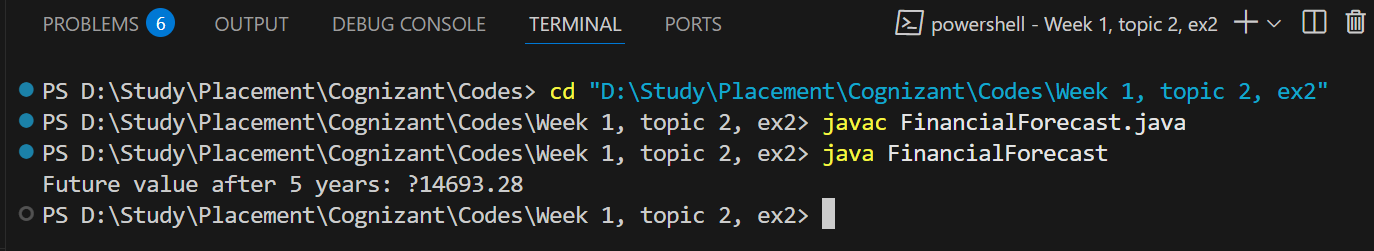
        int years=5;

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

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

    }

}

**Output:**

**4. Analysis:**

#### **Time Complexity:**

Since the function calls itself once per year (n times), the time complexity is:

**T(n) = T(n - 1) + O(1)**  
So, **Time Complexity = O(n)**

This means the execution time increases linearly with the number of periods (years).

### Optimization:

### If we calculate the power of (1 + r)^n directly using efficient methods like **exponentiation by squaring**, or use **iteration instead of recursion**, we can reduce memory usage and improve speed.

public static double calculateFutureValueIterative(double presentValue, double growthRate, int periods)

{

    for (int i = 0; i < periods; i++)

    {

        presentValue \*= (1 + growthRate);

    }

    return presentValue;

}

This version avoids recursion and is better when working with very large n values to prevent **stack overflow** or unnecessary function call overhead.  
  
Recursion is a powerful technique that simplifies certain problems, especially when they involve repeating patterns. In our case, predicting future values with repeated growth is an ideal use case.

However, while recursion gives clean code, it can become inefficient or risky for large inputs due to memory limits. In such cases, using loops or mathematical power functions is better.

| **Aspect** | **Recursive Approach** | **Iterative Approach** |
| --- | --- | --- |
| Code Readability | Very clean & intuitive | Slightly longer |
| Time Complexity | O(n) | O(n) |
| Space Complexity | O(n) due to call stack | O(1) |
| Best for | Small/medium period forecasts | Large period, performance-critical apps |