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

**Big O Notation :** Big O describes how an algorithm’s time or space grows with input size n.

|  |  |  |
| --- | --- | --- |
| **Notation** | **Description** | **Example** |
| O(1) | Constant time | Accessing an array element |
| O(log n) | Logarithmic time | Binary search |
| O(n) | Linear time | Linear search |
| O(n log n) | Efficient sorting | Merge sort, quick sort |
| O(n²) | Quadratic time | Nested loops |

//Product.java

package EcommerceSearch;

public class Product {

    int productId;

    String productName;

    String category;

    public Product(int id, String name, String cat) {

        this.productId = id;

        this.productName = name;

        this.category = cat;

    }

    public String toString() {

        return "ID: " + productId + ", Name: " + productName + ", Category: " + category;

    }

}

//SearchUtil.java

package EcommerceSearch;

import java.util.Arrays;

import java.util.Comparator;

public class SearchUtil {

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

        for (Product p : products) {

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

                return p;

            }

        }

        return null;

    }

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

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

        while (left <= right) {

            int mid = (left + right) / 2;

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

            if (compare == 0) return products[mid];

            else if (compare < 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()));

    }

}

//EcommerceSearchTest.java

package EcommerceSearch;

public class EcommerceSearchTest {

    public static void main(String[] args) {

        Product[] products = {

            new Product(1, "Laptop", "Electronics"),

            new Product(2, "Shampoo", "Beauty"),

            new Product(3, "Phone", "Electronics"),

            new Product(4, "Tablet", "Electronics")

        };

        System.out.println("Linear Search for 'Phone':");

        Product result1 = SearchUtil.linearSearch(products, "Phone");

        System.out.println(result1);

        System.out.println("\nBinary Search for 'Phone':");

        SearchUtil.sortByName(products);

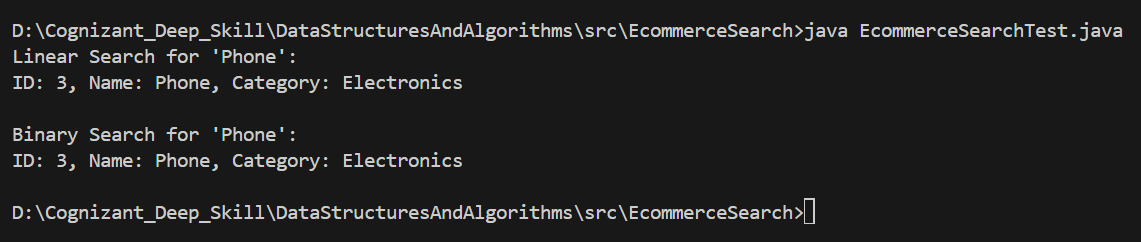
        Product result2 = SearchUtil.binarySearch(products, "Phone");

        System.out.println(result2);

    }

}

OUTPUT:



**Time Complexity:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| **Linear** | O(1) | O(N) | O(N) |
| **Binary** | O(1) | O(log N) | O(log N) |

**Suitability:**

I use **Linear Search** for:

* + Small product list
  + Unsorted data
  + Few searches

I use **Binary Search** for:

* + Large, sorted product catalog
  + Frequent searches
  + Better performance for scale

**Exercise 7: Financial Forecasting**

**Recursion:**

Recursion is a programming technique where a method calls **itself** to solve a smaller version of the problem until a base condition is met.

**Usage of Recursion:**

* Breaks complex problems into smaller ones
* Cleaner and elegant for repetitive, divide-and-conquer tasks

**Goal:**

To predict future value FV using **compound growth**:

Where:

* PV = Present Value
* r = Growth Rate (e.g., 0.05 for 5%)
* n = Number of years

**Program:**

//FinancialForecast.java

package RecursiveAlgorithm;

public class FinancialForecast {

    public static double forecast(double pv, double rate, int years) {

        if (years == 0) {

            return pv;

        }

        return forecast(pv, rate, years - 1) \* (1 + rate);

    }

    public static void main(String[] args) {

        double pv = 1000.0;

        double rate = 0.05;

        int years = 10;

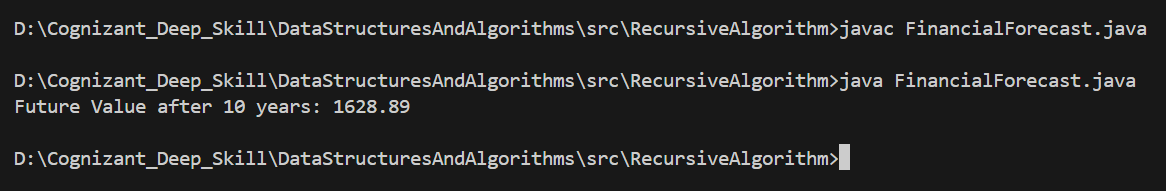
        double fv = forecast(pv, rate, years);

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

    }

}

OUTPUT:



**Time Complexity**

|  |  |
| --- | --- |
| **Operation** | **Complexity** |
| Recursive Forecast | O(n) |

**Optimization Techniques**

**Issue:** Recursive calls recompute values at each step. For large n, it becomes inefficient.

**Solution:** We can use Iteration or Memoization.

* **Time Complexity**: O(n)
* **Space Complexity**: O(1) → better than recursive O(n) stack usage