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

**Big O notation** describes the upper bound of an algorithm’s running time in terms of input size n. It tells you how the performance of your algorithm scales.

| **Big O** | **Description** | **Example** |
| --- | --- | --- |
| O(1) | Constant time | Accessing array element |
| O(n) | Linear time | Linear search |
| O(log n) | Logarithmic time | Binary search |
| O(n²) | Quadratic time | Nested loops |

**Best, Average & Worst Case:**

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| **Linear** | O(1) | O(n/2) ≈ O(n) | O(n) |
| **Binary** | O(1) | O(log n) | O(log n) |

**CODE:**

package week1.dsa;

import java.util.Arrays;

import java.util.Comparator;

class Product{

    int pid;

    String pname;

    String category;

    public Product(int pid, String pname, String category){

        this.pid = pid;

        this.pname = pname;

        this.category = category;

    }

    @Override

    public String toString(){

        return ("[" + this.pid + "] " + this.pname + " (" + this.category + ")");

    }

}

class SearchFunction{

    public static Product LinearSearch(Product[] products, String target\_name){

        for(Product p : products){

            if(p.pname.equalsIgnoreCase(target\_name)){

                return p;

            }

        }

        return null;

    }

    public static Product BinarySearch(Product[] products, int target\_id){

        int low = 0;

        int high = products.length - 1;

        while(low <= high){

            int mid = (low + high) / 2;

            if(products[mid].pid == target\_id){

                return products[mid];

            }

            else if(products[mid].pid < target\_id){

                low = mid + 1;

            }

            else{

                high = mid - 1;

            }

        }

        return null;

    }

    public static void sortProducts(Product[] products) {

        Arrays.sort(products, Comparator.comparingInt(p -> p.pid));

    }

}

public class E\_commercePlatformSearchFunction {

    public static void main(String[] args) {

        Product[] products = {

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

            new Product(102, "Mobile", "Electronics"),

            new Product(103, "Watch", "Accessories"),

            new Product(104, "Shirt", "Clothing"),

            new Product(105, "Book", "Education"),

            new Product(106, "Shoes", "Footwear")

        };

        Product res1 = SearchFunction.LinearSearch(products, "Laptop");

        System.out.println("Linear Search Result: " + (res1 != null ? res1 : "Not Found"));

        SearchFunction.sortProducts(products);

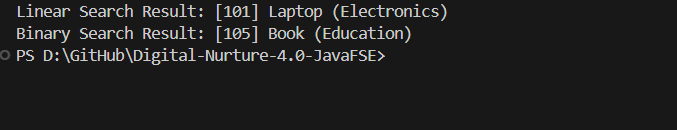
        Product res2 = SearchFunction.BinarySearch(products, 105);

        System.out.println("Binary Search Result: " + (res2 != null ? res2 : "Not Found"));

    }

}

**OUTPUT:**

****

**Analysis:**

**Time Complexity Comparison:**

In terms of large datasets Binary Search is better than Linear Search.

| **Algorithm** | **Time Complexity** | **When to Use** |
| --- | --- | --- |
| Linear Search | O(n) | Small or unsorted datasets |
| Binary Search | O(log n) | Large and sorted datasets |

**Exercise 7: Financial Forecasting**

**Recursion** is a programming technique where a method **calls itself** to solve a smaller instance of the same problem.

**CODE:**

package week1.dsa;

public class FinancialForecasting {

    public static double forecast(double initial\_amt, double rate, int years) {

        if (years == 0) {

            return initial\_amt;

        } else {

            return forecast(initial\_amt, rate, years - 1) \* (1 + rate); //recursion

        }

    }

    public static void main(String[] args) {

        double p\_amt = 1000.0; // Initial amount

        double growthRate = 0.10;  // 10% per year

        int years = 5;

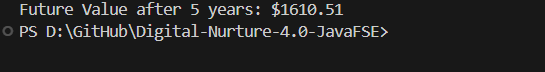
        double futureValue = forecast(p\_amt, growthRate, years);

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

    }

}

**OUTPUT:**

****

**Analyse:**

**Time Complexity:**

* Each call reduces years by 1.
* So, **T(n) = O(n)** — **linear time**.

**Optimization:**

Recursion can be inefficient in deep calls. Use **Memoization** or convert to **iteration** if performance is critical.