**Algorithms and Data Structures**

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

**Big O Notation**

* Asymptotic Analysis is the big idea that handles the above issues in analyzing algorithms.
* Big O is a way to express the upper boundof an algorithm’s time or space complexity.
* **The time complexity** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
* The amount of memory required by the algorithm to solve given problem is called **space complexity** of the algorithm.
* It helps in estimating **scalability and performance**, especially in worst-case scenarios.
* It helps **predict performance** without needing to run the program.
* It allows you to **compare algorithms** independently of hardware or programming language.

**Big O notation is important for several reasons:**

* Big O Notation is important because it helps analyze the efficiency of algorithms.
* It provides a way to describe how the**runtime**or **space requirements** of an algorithm grow as the input size increases.
* Allows programmers to compare different algorithms and choose the most efficient one for a specific problem.
* Helps in understanding the scalability of algorithms and predicting how they will perform as the input size grows.
* Enables developers to optimize code and improve overall performance.

**How it works?**

Given two functions**f(n)** and **g(n)**,

we say that**f(n)** is**O(g(n))** if there exist constants**c > 0** and **n0 >= 0** such that

**f(n) <= c\*g(n)** for all **n >=n0**.

* **O(1) :** **Constant Time**

The algorithm does the same number of steps no matter how large the input is. Example: Array access by index.

* **O(log n) : Logarithmic Time**

Each step eliminates half of the problem, so it takes fewer steps as it grows.

Example: Binary search

* **O(n):** **Linear Time**

The number of steps grows directly in proportion to the input size.

Example: Linear search, single loop

* **O(n²): Quadratic Time**

Steps grow proportionally to the square of the input size.

Example: Nested loops, bubble sort

* **O(2ⁿ): Exponential Time**

The steps double with every additional input.

Example: Recursive Fibonacci, permutations

Best Case Analysis

* In the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed.
* The search finds the item immediately or with the fewest comparisons.
* Linear Search: Best Case is O(1), if item is at the first position.
* Binary Search: Best Case is O(1), if item is at the middle index on the first try.

Average Case Analysis

* In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs.
* The search takes a typical or expected number of comparisons.
* Linear Search: Average Case is O(n), if item is somewhere in the middle.
* Binary Search: Average Case is O(log n), if Item is found after a few splits.

Worst Case Analysis

* In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed.
* The search takes the maximum possible comparisons or fails completely.
* Linear Search: Best Case is O(n), if item is the last in the list or not present at all.
* Binary Search: Best Case is O(log n), if item is not found or is at the deepest level of search.

**CODE**

import java.util.Arrays;

import java.util.Comparator;

class Product {

    private int productId;

    private String productName;

    private String category;

    public Product(int productId, String productName, String category) {

        this.productId = productId;

        this.productName = productName;

        this.category = category;

    }

    public int getProductId() {

        return productId;

    }

    public String getProductName() {

        return productName;

    }

    public String getCategory() {

        return category;

    }

    @Override

    public String toString() {

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

    }

}

class SearchUtility {

    // Linear Search by product name

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

        for (Product product : products) {

            if (product.getProductName().equalsIgnoreCase(targetName)) {

                return product;

            }

        }

        return null;

    }

    // Binary Search by product name

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

        int left = 0;

        int right = products.length - 1;

        while (left <= right) {

            int mid = (left + right) / 2;

            int comparison = products[mid].getProductName().compareToIgnoreCase(targetName);

            if (comparison == 0) {

                return products[mid];

            } else if (comparison < 0) {

                left = mid + 1;

            } else {

                right = mid - 1; } }

     return null;

    }

    // sort products by name

    public static void sortProductsByName(Product[] products) {

        Arrays.sort(products, Comparator.comparing(Product::getProductName, String.CASE\_INSENSITIVE\_ORDER));

    }

}

// main class

public class Main {

    public static void main(String[] args) {

        Product[] products = {

            new Product(101, "Shoes", "Footwear"),

            new Product(102, "T-Shirt", "Clothing"),

            new Product(103, "Laptop", "Electronics"),

            new Product(104, "Backpack", "Accessories"),

            new Product(105, "Watch", "Accessories")

        };

        // Linear Search Test

        Product foundLinear = SearchUtility.linearSearch(products, "Laptop");

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

        // Sorting array by name

        SearchUtility.sortProductsByName(products);

        // Binary Search Test

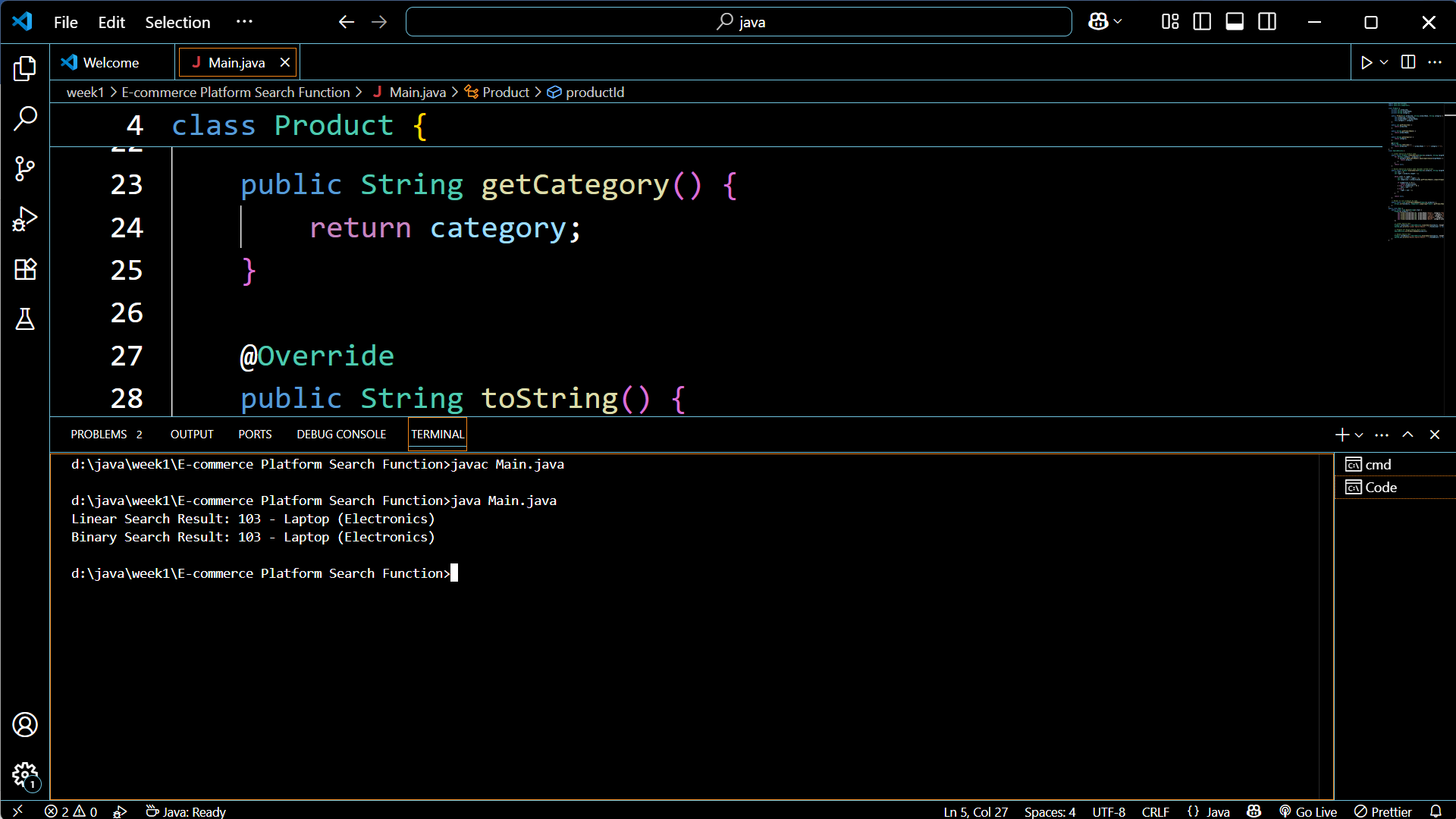
        Product foundBinary = SearchUtility.binarySearch(products, "Laptop");

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

    }

}

**OUPUT**



**TIME COMPLEXITY**

* LINEAR SEARCH IS O(n).
* BINARY SEARCH IS O(log n).

**Which algorithm is more suitable for our platform and why?**

* For searching products on an e-commerce website, binary search is generally preferred over linear search, especially for large product catalogs.
* Binary search is preferred in e-commerce if searching on **sorted** or indexed fields — which is common for high-performance search systems.

Advantages of Binary Search:

* Highly efficient for large, sorted datasets.
* Time complexity is O(log n), which is significantly faster than linear search for larger datasets.
* Very scalable, it can handle millions of products efficiently.
* Also by using Binary Search, it results in faster search results lead to a better user experience, as customers can quickly find what they're looking for.

Disadvantage of Linear Search for this purpose:

* Inefficient for large datasets, as it needs to check every item in the worst case.
* Time complexity: O(n) , meaning time grows linearly with the number of products.
* Inefficient when the product is not at the beginning.

**Exercise 7: Financial Forecasting**

**Recursion**

* The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.
* It works by:

1. Breaking a big problem into smaller subproblems of the same type.
2. Solving the smallest version (called the base case).
3. Combining results from the smaller versions to get the final answer.

**Need of Recursion?**

* Recursion helps in logic building. Recursive thinking helps in solving complex problems by breaking them into smaller subproblems.
* Recursive solutions work as a a basis for Dynamic Programming and Divide and Conquer algorithms.
* Certain problems can be solved quite easily using recursion like Tower of Hanoi, Tree Traversal etc.

**How Recursion Simplifies Problems?**

**1.** **Reduces Complex Code**

* Recursive solutions are often shorter and clearer than loops, especially for problems with repetitive patterns, such as:
  + Tree/graph traversal
  + Mathematical sequences
  + Nested data structures (e.g., file systems, XML)

**2. Natural Fit for Divide-and-Conquer**

* Problems like binary search, merge sort, or quick sort divide the task into sub-tasks — recursion handles this naturally.

**3. Elegant for Backtracking**

* Used in puzzles and decision trees (e.g., Sudoku, pathfinding, permutations).

**CODE**

public class ForecastingTool {

    // Recursive method to compute future value

    public static double predictFutureValue(double initialValue, double growthRate, int years) {

        if (years == 0) {

            return initialValue; // Base case

        }

        return predictFutureValue(initialValue, growthRate, years - 1) \* (1 + growthRate);

    }

    public static void main(String[] args) {

        double initialValue = 10000.0; // Starting capital

        double growthRate = 0.05; // 5% annual growth

        int years = 5; // Forecast for 5 years

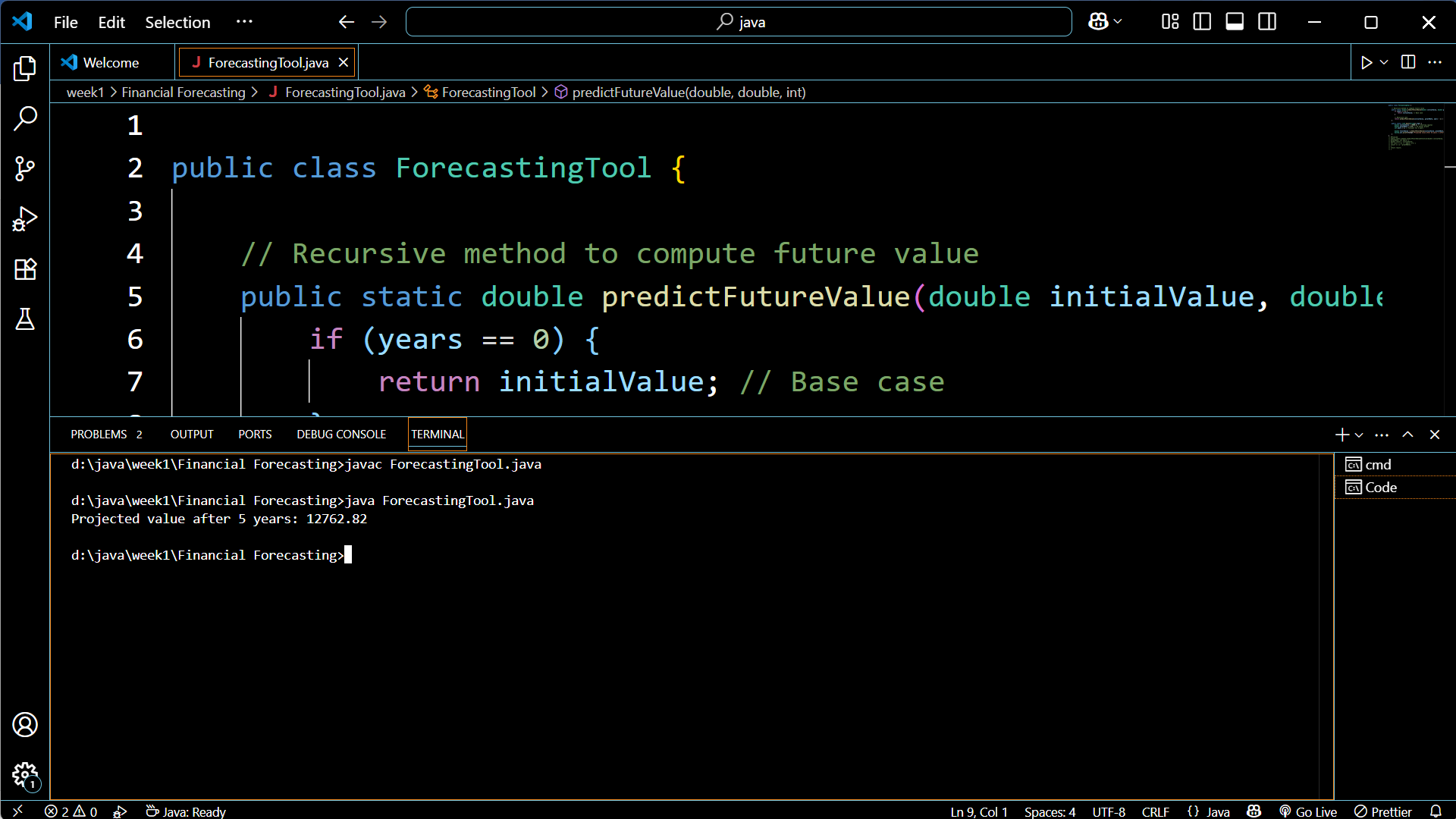
        double futureValue = predictFutureValue(initialValue, growthRate, years);

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

    }

}

**OUTPUT**



**Time Complexity**

Each recursive call reduces years by 1 until it hits 0:

* As each recursive call does **one multiplication** and **one recursive call**.

**Time Complexity:** O(n), where n = number of years

* As each call is added to the call stack.

**Space Complexity:** O(n) due to call stack usage

**Optimization**

**1. Top-Down Approach (Memoization):**

In the top-down approach, also known as memoization, we keep the solution recursive and add a memoization table to avoid repeated calls of same subproblems.

* Before making any recursive call, we first check if the memoization table already has solution for it.
* After the recursive call is over, we store the solution in the memoization table.

**2. Bottom-Up Approach (Tabulation):**

In the bottom-up approach, also known as tabulation, we start with the smallest subproblems and gradually build up to the final solution.

* We write an iterative solution (avoid recursion overhead) and build the solution in bottom-up manner.
* We use a dp table where we first fill the solution for base cases and then fill the remaining entries of the table using recursive formula.
* We only use recursive formula on table entries and do not make recursive calls.

**We can optimize are code by changing predictFutureValue as below**

public static double predictFutureValue(double initialValue, double growthRate, int years) {

double result = initialValue;

for (int i = 0; i < years; i++) {

result \*= (1 + growthRate);

}

return result;

}