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

**Understanding Asymptotic Notation**

Big O notation tells us how fast or slow an algorithm is as the data grows. It helps compare algorithms without needing exact times. For example, O(n) means time grows with data size, while O(log n) (like binary search) is much faster.

**CODE:**

**Product Class Code:**

package week\_1\_Data\_Structure\_Algorithm\_E\_Commerce;

import java.util.\*;

//Product Class

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 String getProductName() {

return productName;

}

*@Override*

public String toString() {

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

}

}

**Product Search Class Code:**

package week\_1\_Data\_Structure\_Algorithm\_E\_Commerce;

import java.util.Arrays;

import java.util.Comparator;

import java.util.Scanner;

public class ProductSearch {

// Linear Search

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

for (Product product : products) {

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

return product;

}

}

return null;

}

// Binary Search (Assumes sorted array by product name)

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

String midName = products[mid].getProductName().toLowerCase();

int compare = midName.compareTo(targetName.toLowerCase());

if (compare == 0) {

return products[mid];

} else if (compare < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

// Main Method

public static void main(String[] args) {

Scanner scanner = new Scanner(System.***in***);

// Sample Products

Product[] products = {

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

new Product(2, "Phone", "Electronics"),

new Product(3, "Shoes", "Apparel"),

new Product(4, "Watch", "Accessories"),

new Product(5, "Tablet", "Electronics"),

new Product(6, "Camera", "Electronics")

};

System.***out***.print("Enter product name to search: ");

String input = scanner.nextLine();

// ---- Linear Search ----

long startLinear = System.*nanoTime*();

Product resultLinear = *linearSearch*(products, input);

long endLinear = System.*nanoTime*();

System.***out***.println("\nLinear Search:");

if (resultLinear != null)

System.***out***.println("Found: " + resultLinear);

else

System.***out***.println("Product not found.");

System.***out***.println("Time Complexity: O(n)");

System.***out***.println("Execution Time: " + (endLinear - startLinear) + " ns");

// ---- Binary Search ----

Arrays.*sort*(products, Comparator.*comparing*(p -> p.getProductName().toLowerCase()));

long startBinary = System.*nanoTime*();

Product resultBinary = *binarySearch*(products, input);

long endBinary = System.*nanoTime*();

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

if (resultBinary != null)

System.***out***.println("Found: " + resultBinary);

else

System.***out***.println("Product not found.");

System.***out***.println("Time Complexity: O(log n)");

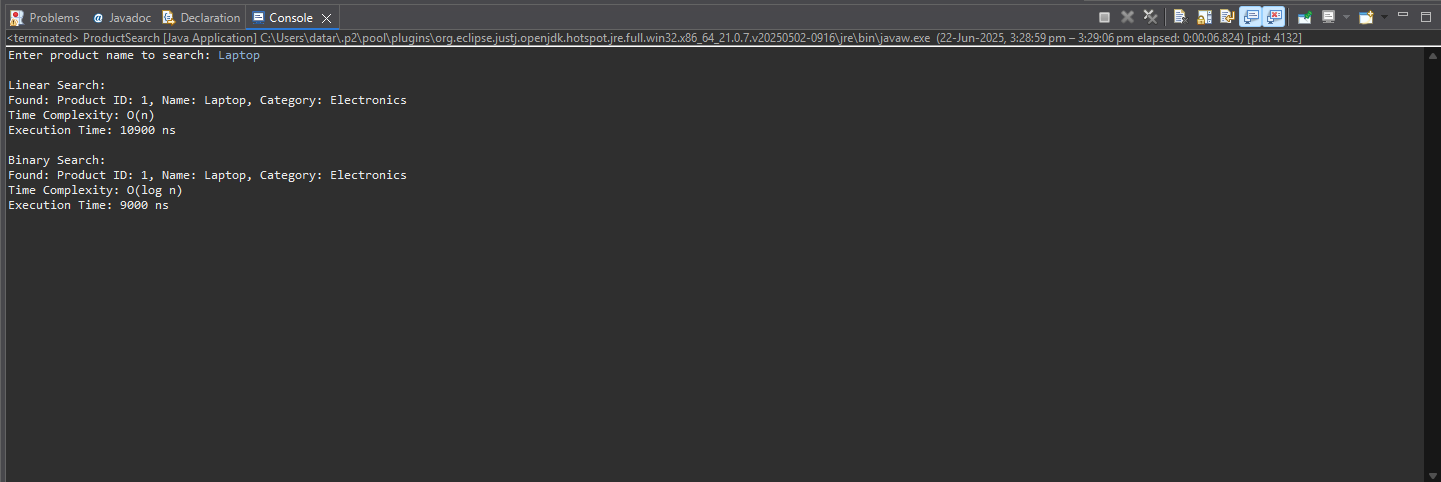
System.***out***.println("Execution Time: " + (endBinary - startBinary) + " ns");

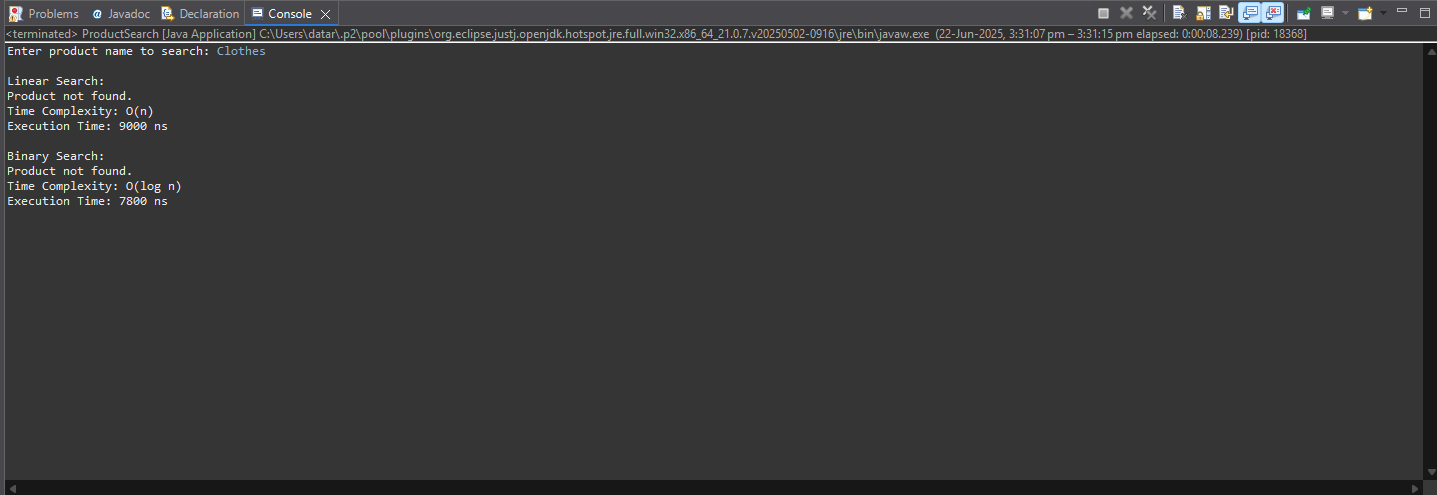
scanner.close();

}

}

**OUTPUTS:**





**Analysis:**

* **Question 1: Compare the time complexity of linear and binary search algorithms.**
* Linear Search goes through each item one by one. It's simple and works on unsorted data, but slow for large datasets.
* Binary Search repeatedly divides a sorted list in half, making it much faster for large data.
* **Discuss which algorithm is more suitable for your platform and why.**

For an e-commerce platform, where users may search through thousands of products, binary search is more suitable because of its O(log n) time complexity. It performs much faster, especially as the product list grows.

However, binary search requires data to be sorted. If your platform maintains a sorted product list (by name or ID), binary search is ideal. If not, and data is frequently changing or unsorted, linear search might be simpler—but less efficient.

In most cases, binary search is better for performance in large-scale platforms.

| **Case** | **Linear Search** | **When It Happens** | **Binary Search** | **When It Happens** |
| --- | --- | --- | --- | --- |
| **Best Case** | O(1) | Target is the **first element** in the list | O(1) | Target is the **middle element** of the sorted list |
| **Average Case** | O(n) | Target is **somewhere in the middle** | O(log n) | Target is in the list, but not near the middle |
| **Worst Case** | O(n) | Target is the **last element** or **not found** | O(log n) | Target is **not in the list** or far from middle |

**Exercise 7: Financial Forecasting**

**Understanding Recursive Algorithm**

Recursion is when a function calls itself to solve a smaller piece of the same problem. It continues doing this until it reaches a basic condition called the base case, which stops the recursion.

Recursion is powerful because it lets you solve complex problems by breaking them into smaller, simpler ones. It’s especially helpful in problems that have a repetitive or nested structure, like:

* Calculating factorials
* Navigating folders or trees
* Solving mathematical sequences
* Forecasting future values

FutureValue(n) = FutureValue(n − 1) × (1 + growthRate)

**CODE:**

**Forecast Class Code**

package week\_1\_Data\_Structure\_Algorithm\_Financial\_Forecasting;

import java.util.Scanner;

public class Forecast {

// Recursive method

public static double futureValueRecursive(double presentValue, double rate, int years) {

if (years == 0) {

return presentValue;

}

return *futureValueRecursive*(presentValue, rate, years - 1) \* (1 + rate);

}

// Iterative method

public static double futureValueIterative(double presentValue, double rate, int years) {

double result = presentValue;

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

result \*= (1 + rate);

}

return result;

}

// Main method

public static void main(String[] args) {

Scanner scanner = new Scanner(System.***in***);

// Take user input

System.***out***.print("Enter the present value (e.g., 1000): ");

double initialValue = scanner.nextDouble();

System.***out***.print("Enter the annual growth rate (as % e.g., 5 for 5%): ");

double annualRate = scanner.nextDouble() / 100.0;

System.***out***.print("Enter the number of years: ");

int years = scanner.nextInt();

// Calculations

double recursiveResult = *futureValueRecursive*(initialValue, annualRate, years);

double iterativeResult = *futureValueIterative*(initialValue, annualRate, years);

// Output

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

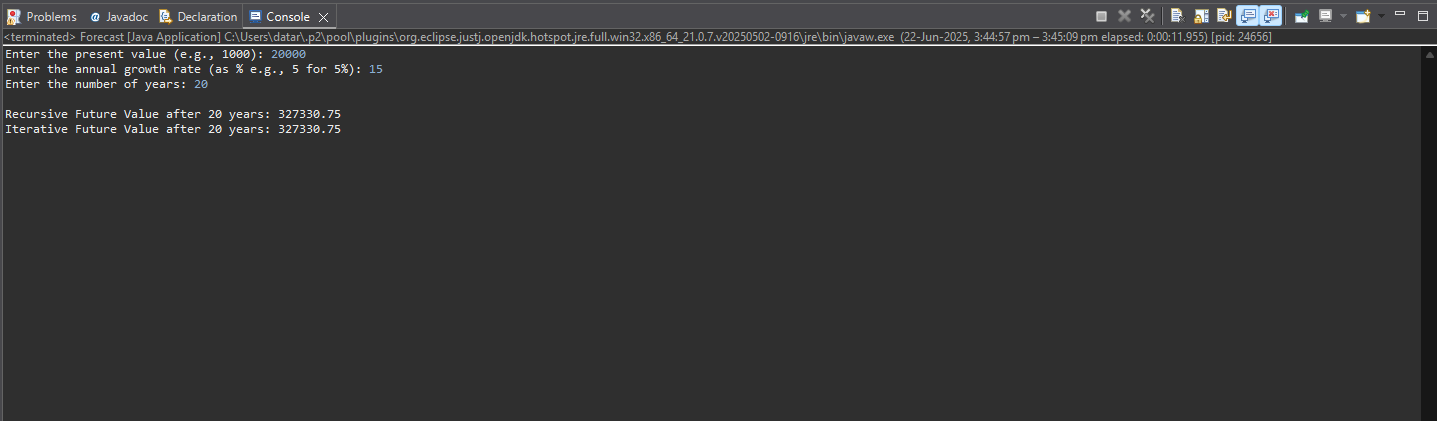
System.***out***.printf("Iterative Future Value after %d years: %.2f\n", years, iterativeResult);

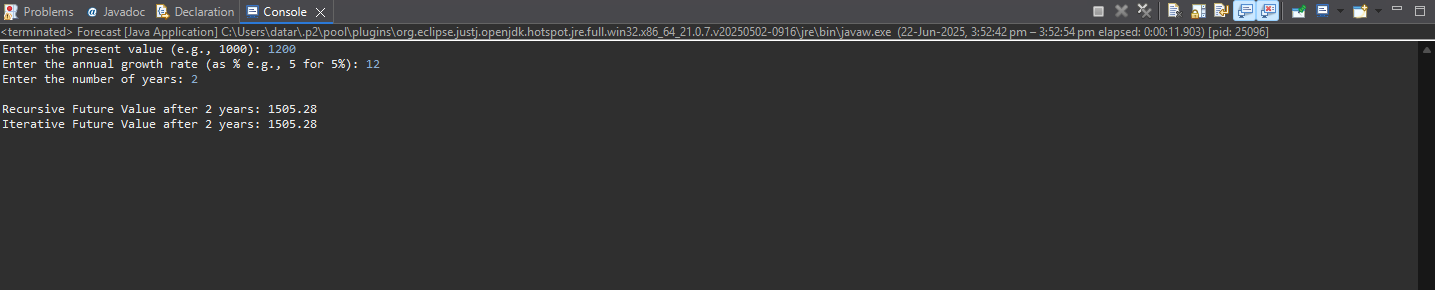
scanner.close();

}

}

**OUTPUTS:**





**Analysis:**

* **Question 1 : Discuss the time complexity of your recursive algorithm**

For the futureValueRecursive method:

public static double futureValueRecursive(double presentValue, double rate, int years) {

if (years == 0) return presentValue;

return futureValueRecursive(presentValue, rate, years - 1) \* (1 + rate);

}

This function calls itself once for each year, reducing years by 1 until it reaches 0.  
So, the time complexity is:

O(n) — where n is the number of years.

* **Recursive Time Complexity:** O(n)
  + Each recursive call reduces the years by 1 until it reaches 0.
* **Space Complexity:** O(n)
  + Due to the call stack for each recursive invocation.
* **Question 2 : Explain how to optimize the recursive solution to avoid excessive computation**.

Although the current recursion is linear and not too expensive, for **larger years values** or multiple forecasts, prefer an **iterative** or **memoized** version.

* Iterative Version (Optimized):

public static double futureValueIterative(double currentValue, double growthRate, int years) {

double result = currentValue;

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

result \*= (1 + growthRate);

}

return result;

}

* Memoized Version (if multiple overlapping forecasts):

import java.util.HashMap;

public class FinancialForecastMemo {

private static HashMap<Integer, Double> memo = new HashMap<>();

public static double futureValue(double currentValue, double growthRate, int years) {

if (years == 0) return currentValue;

if (memo.containsKey(years)) return memo.get(years);

double result = futureValue(currentValue, growthRate, years - 1) \* (1 + growthRate);

memo.put(years, result);

return result;

}

}