**01 : E-commerce Platform Search Function**

**Big O Notation:**

**Big O notation describes how the runtime or space requirements of an algorithm grow relative to the input size n.**

| **Notation** | **Name** | **Meaning** |
| --- | --- | --- |
| **O(1)** | **Constant Time** | **Execution time doesn’t depend on input size** |
| **O(n)** | **Linear Time** | **Execution time grows linearly with input** |
| **O(log n)** | **Logarithmic** | **Grows slowly even for large input** |
| **O(n²)** | **Quadratic** | **Time grows quickly with input size** |

**🔹 Best, Average, and Worst-Case Scenarios (for searching):**

* **Linear Search (Unsorted Array):**
  + **Best: O(1) → First element matches**
  + **Average: O(n/2) ≈ O(n)**
  + **Worst: O(n) → Last element or not found**
* **Binary Search (Sorted Array):**
  + **Best: O(1) → Middle element is the target**
  + **Average: O(log n)**
  + **Worst: O(log n)**

**CODE:**

Product class:

class Product {

int productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

*@Override*

public String toString() {

return "[" + productId + ", " + productName + ", " + category + "]";

}

}

Product Search Class:

**import java.util.Arrays;**

**import java.util.Comparator;**

**public class ProductSearch {**

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

**for (Product product : products) {**

**if (product.productName.equalsIgnoreCase(name)) {**

**return product; }**

**}**

**return null; }**

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

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

**while (left <= right) {**

**int mid = left + (right - left) / 2;**

**int cmp = products[mid].productName.compareToIgnoreCase(name);**

**if (cmp == 0) return products[mid];**

**else if (cmp < 0) left = mid + 1;**

**else right = mid - 1;**

**}**

**return null;**

**}**

**public static void main(String[] args) {**

**Product[] products = {**

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

**new Product(102, "Shampoo", "Personal Care"),**

**new Product(103, "Shoes", "Footwear"),**

**new Product(104, "Phone", "Electronics"),**

**new Product(105, "Book", "Stationery")**

**};**

**System.*out*.println("Linear Search: " + *linearSearch*(products, "Phone"));**

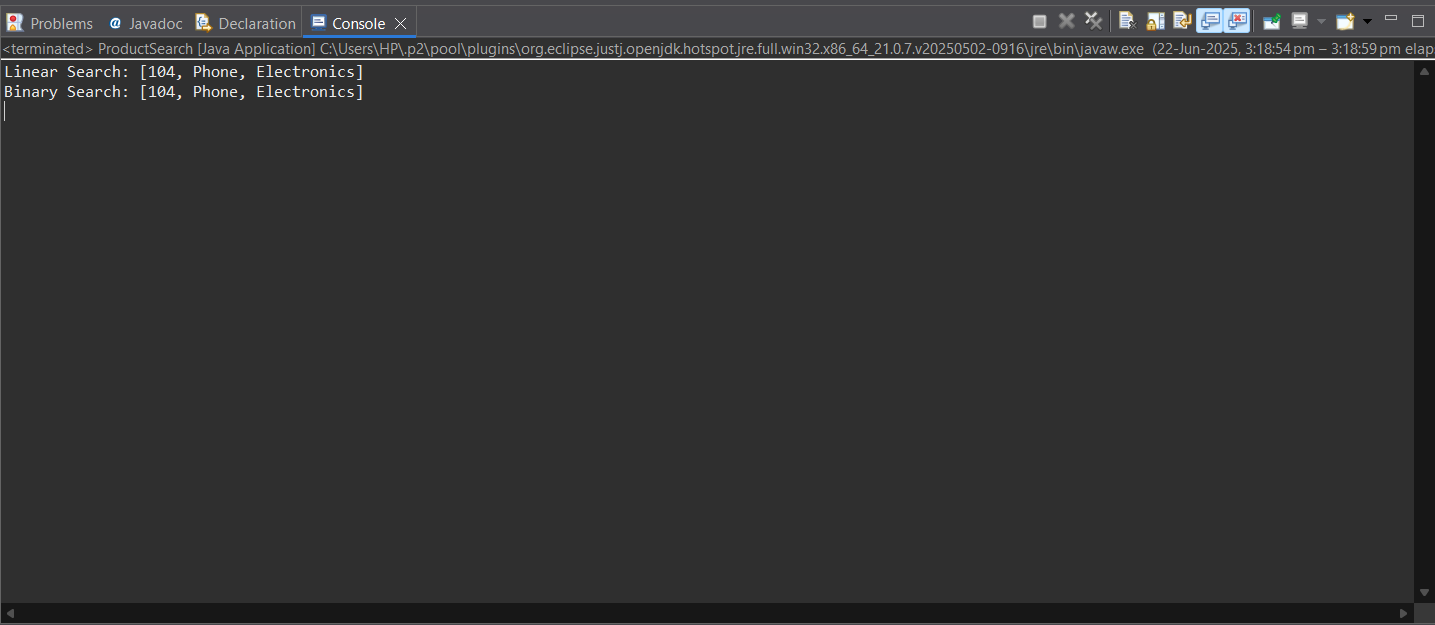
**Arrays.*sort*(products, Comparator.*comparing*(p -> p.productName.toLowerCase()));**

**System.*out*.println("Binary Search: " + *binarySearch*(products, "Phone"));**

**}**

**}**

**OUTPUT:**

****

**ANALYSIS:**

* **Question 1:** Compare the time complexity of linear and binary search algorithms.

|  |  |  |  |
| --- | --- | --- | --- |
| **ALGORITHM** | **Time Complexity** | **Space Complexity** | **Requirement** |
| Linear Search | O(n) | O(1) | No sorting required |
| Binary Search | |  |  | | --- | --- | |  | O(log n) | | |  |  |  | | --- | --- | --- | |  |  | O(1) | | Array must be sorted |

* Use Linear Search if:
  + The data is unsorted.
  + You have few elements or frequent insertions.
* Use Binary Search if:
  + The data is sorted
  + You want fast search on large datasets.
  + The list is static or infrequently modified.
* **Question 2**: Discuss which algorithm is more suitable for your platform and why.

For an e-commerce platform with large product data, I will prefer Binary Search on sorted collections to optimize performance.

**02: Financial Forecasting**

* **What is Recursion?**

Recursion is when a method **calls itself** to solve smaller subproblems. It simplifies problems that have a **repeating pattern** or that can be **broken into smaller identical problems**.

For financial forecasting, if we know the value at year n-1, we can calculate value at year n using:

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

* **CODE :**  
    
    
  public class FinancialForecast {

// Recursive method to calculate future value

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

if (years == 0) {

return currentValue;

}

return futureValue(currentValue, growthRate, years - 1) \* (1 + growthRate);

}

public static void main(String[] args) {

double currentValue = 1000.0; // Starting investment

double growthRate = 0.05; // 5% annual growth

int years = 5; // Predict for 5 years

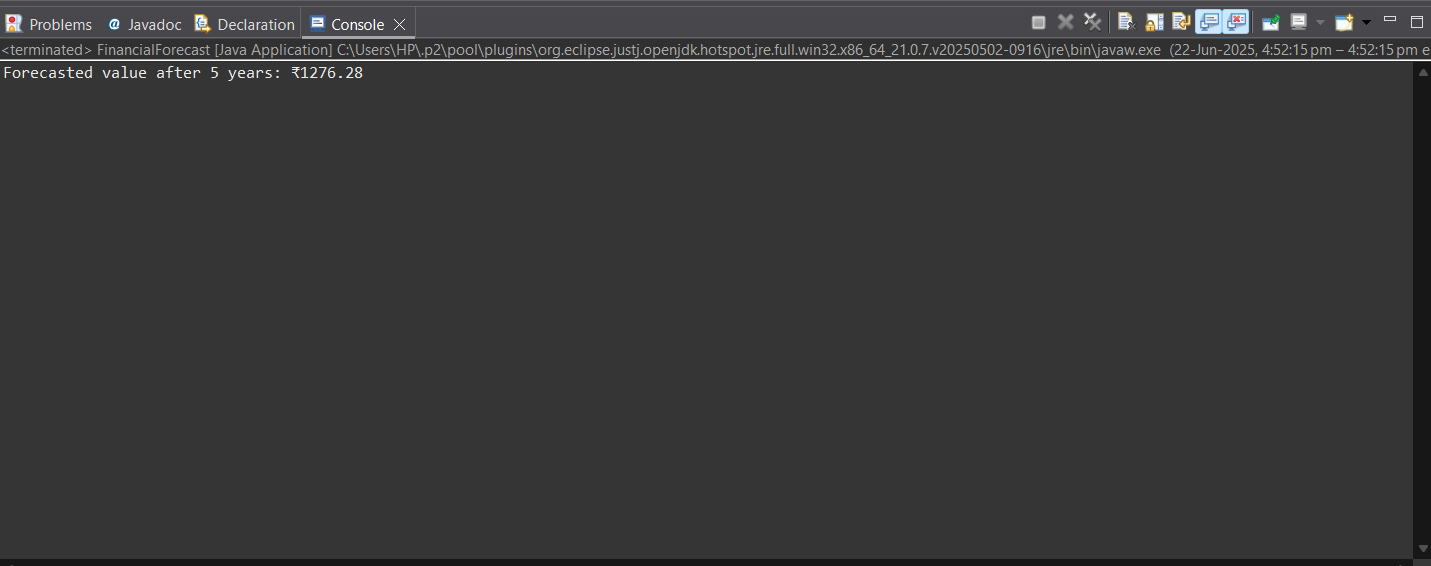
double forecastedValue = futureValue(currentValue, growthRate, years);

System.out.printf("Forecasted value after %d years: ₹%.2f%n", years, forecastedValue);

}

}

**OUTPUT:**

****

**ANALYSIS:**

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

**Time Complexity:**

* **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.

**🔹 Optimization: Avoiding 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.

* + **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;

}

}