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

**Big O Notation**

* **Big O expresses the worst-case growth rate of an algorithm.**
* **It helps us evaluate efficiency and scalability.**

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

**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 String getProductName() {

return productName;

}

@Override

public String toString() {

return "Product{" +

"ID=" + productId +

", Name='" + productName + '\'' +

", Category='" + category + '\'' +

'}';

}

}

public class ProductSearchDemo {

// 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

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();

int compare = midName.compareToIgnoreCase(targetName);

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) {

// Sample Product Array

Product[] products = {

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

new Product(102, "Shoes", "Footwear"),

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

new Product(104, "Camera", "Electronics")

};

// Linear Search

System.out.println("=== Linear Search ===");

Product result1 = linearSearch(products, "Watch");

if (result1 != null) {

System.out.println("Product found: " + result1);

} else {

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

}

// Sort array by productName for Binary Search

Arrays.sort(products, Comparator.comparing(Product::getProductName));

// Binary Search

System.out.println("\n=== Binary Search ===");

Product result2 = binarySearch(products, "Watch");

if (result2 != null) {

System.out.println("Product found: " + result2);

} else {

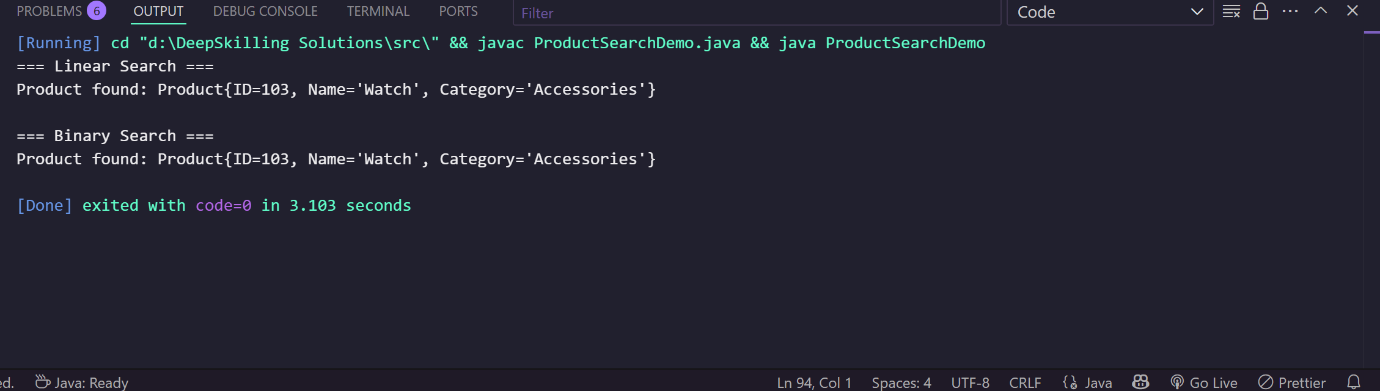
System.out.println("Product not found");

}

}

}

**Output:**

****

| **Feature** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| Time Complexity | O(n) | O(log n) |
| Requires Sorting | No | Yes |
| Suitable for | Small / unsorted data | Large / sorted datasets |
| Performance | Slower with large input | Faster with large datasets |

For a **large e-commerce platform**, binary search is **more efficient** if:

* The product list is sorted (or indexed).
* Search operations are frequent.

We should use **Binary Search** for better performance in production

**Exercise 7: Financial Forecasting**

**What is Recursion?**

* **Recursion** is a technique where a method **calls itself** to solve a problem.
* Each recursive call solves a **smaller subproblem**, and the **base case** ends the recursion.

**Why Use Recursion in Forecasting?**

* Forecasting often involves **repeating a growth pattern** over time.
* Recursion is natural for problems like compound interest or exponential growth where **future value depends on previous values**.

We'll calculate the **future value** based on:

* initialAmount: starting capital
* growthRate: annual growth rate (e.g., 5% = 0.05)
* years: number of years to forecast

Formula: futureValue(n) = futureValue(n - 1) \* (1 + growthRate)

**Code:**

public class FinancialForecast {

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

if (years == 0) {

return initialAmount;

}

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

}

public static double futureValueMemo(double initialAmount, double growthRate, int years, Double[] memo) {

if (years == 0) {

return initialAmount;

}

if (memo[years] != null) {

return memo[years];

}

memo[years] = futureValueMemo(initialAmount, growthRate, years - 1, memo) \* (1 + growthRate);

return memo[years];

}

public static void main(String[] args) {

double initialAmount = 10000.0;

double growthRate = 0.05;

int years = 10;

double valueRecursive = futureValue(initialAmount, growthRate, years);

System.out.printf("Future Value (Recursive): ₹%.2f\n", valueRecursive);

Double[] memo = new Double[years + 1];

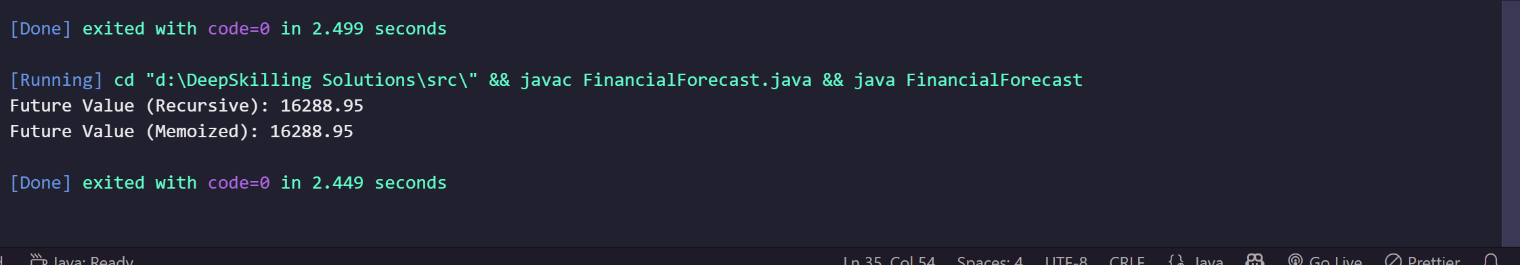
double valueMemo = futureValueMemo(initialAmount, growthRate, years, memo);

System.out.printf("Future Value (Memoized): ₹%.2f\n", valueMemo);

}

}

**Output:**

****

**Analysis**

Time Complexity

* Without Memoization:
  + Each recursive call makes 1 sub-call, so:
  + Time complexity: O(n)
  + Space complexity: O(n) (due to call stack)
* With Memoization:
  + Time complexity: O(n)
  + Space complexity: O(n) (both for memo and stack)

Memoization avoids recomputing previously solved subproblems.