**Week-1 Data Structures and Algorithms**

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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

**1. Understand Asymptotic Notation:**

* Explain Big O notation and how it helps in analyzing algorithms.
* Describe the best, average, and worst-case scenarios for search operations.

**2. Setup:**

* Create a class Product with attributes for searching, such as productId, productName, and category.

**3. Implementation:**

* Implement linear search and binary search algorithms.
* Store products in an array for linear search and a sorted array for binary search.

**4. Analysis:**

* Compare the time complexity of linear and binary search algorithms.
* Discuss which algorithm is more suitable for your platform and why.

**1.Understand Asymptotic Notation**

**Big O Notation**:

Big O Notation describes how the time or space requirement of an algorithm grows as the input size increases. It helps us compare algorithms in terms of their efficiency.

**Best, Average, and Worst Case (for search)**

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Linear Search** | **Binary Search** |
| **Best** | O(1) – First element match | O(1) – Middle element match |
| **Average** | O(n/2) ≈ O(n) | O(log n) |
| **Worst** | O(n) – Not found | O(log n) – Not found |

**2.Setup:**

***Product.java***

package com.EcommercePlatform;

public 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 "Product [ID: " + productId + ", Name: " + productName + ", Category: " + category + "]";

}

}

**3.Implementation:**

***ProductSearch.java***

package com.EcommercePlatform;

import java.util.Arrays;

import java.util.Comparator;

public class ProductSearch {

// Linear Search (unsorted)

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

for (Product product : products) {

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

return product;

}

}

return null;

}

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

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

while (left <= right) {

int mid = (left + right) / 2;

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

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

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

else right = mid - 1;

}

return null;

}

public static void sortProductsByName(Product[] products) {

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

}

}

***SearchTest.java***

package com.EcommercePlatform;

public class SearchTest {

public static void main(String[] args) {

Product[] products = {

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

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

new Product(103, "Shirt", "Clothing"),

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

new Product(105, "Shoes", "Footwear")

};

String searchTarget = "Phone";

System.out.println("=== E-COMMERCE PRODUCT SEARCH SYSTEM ===\n");

// Linear Search

System.out.println("Linear Search for '" + searchTarget + "':");

Product linearResult = ProductSearch.linearSearch(products, searchTarget);

if (linearResult != null) {

System.out.println("Found: " + linearResult.productId + " - " +

linearResult.productName + " (" + linearResult.category + ")");

} else {

System.out.println("Not Found");

}

// Sort before Binary Search

ProductSearch.sortProductsByName(products);

// Binary Search

System.out.println("\nBinary Search for '" + searchTarget + "':");

Product binaryResult = ProductSearch.binarySearch(products, searchTarget);

if (binaryResult != null) {

System.out.println("Found: " + binaryResult.productId + " - " +

binaryResult.productName + " (" + binaryResult.category + ")");

} else {

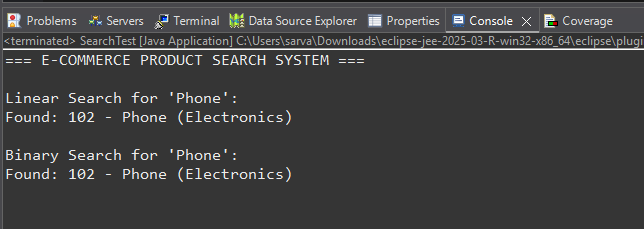
System.out.println("Not Found");

}

}

}

**Output:**



**4. Analysis**

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Description** |
| **Linear Search** | O(n) | Checks each element one by one |
| **Binary Search** | O(log n) | Repeatedly halves the search space |

**Exercise 7:** Financial Forecasting

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**

* Explain the concept of recursion and how it can simplify certain problems.

1. **Setup:**

* Create a method to calculate the future value using a recursive approach.

1. **Implementation:**

* Implement a recursive algorithm to predict future values based on past growth rates.

1. **Analysis:**

* Discuss the time complexity of your recursive algorithm.
* Explain how to optimize the recursive solution to avoid excessive computation.

**Step 1:** Understand Recursive Algorithms

Recursion is when a method calls itself to solve a smaller version of a problem. It continues until it reaches a base case, which stops the recursion.

In forecasting, each year’s value depends on the previous year’s value:

FV(n)=FV(n−1)×(1+r)FV(n) = FV(n-1) \times (1 + r)FV(n)=FV(n−1)×(1+r)

Where:

* FV(n) = future value at year n
* r = annual growth rate

Using recursion, you can define this in code naturally and compactly, rather than using a loop.

**Step 2:** Setup

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

**Step 3:** Implementation

***FinancialForecast.java***

public class FinancialForecast {

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

if (years == 0) {

return initialValue;

} else {

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

}

}

public static void main(String[] args) {

double initialValue = 10000;

double growthRate = 0.1; // 10% annual growth

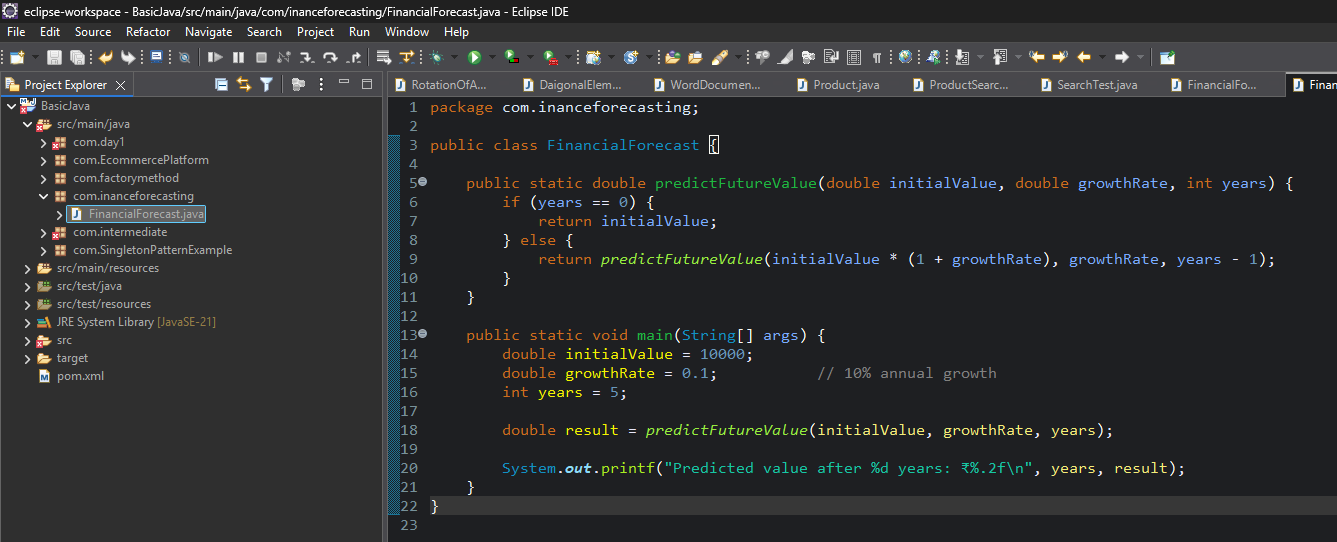
int years = 5;

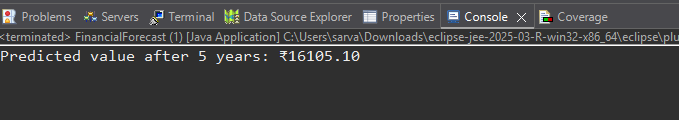
double result = predictFutureValue(initialValue, growthRate, years);

System.out.printf("Predicted value after %d years: ₹%.2f\n", years, result);

}

}



****

**Step 4:** Analysis

Time Complexity :

* Each function call reduces years by 1.
* So if years = n, then:
  + Time complexity = O(n)
  + Space complexity = O(n) (due to call stack)

**Optimization:**

**Convert to Iterative:**

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

double result = initialValue;

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

result \*= (1 + growthRate);

}

return result;

}

**Use Math Formula (Most Efficient):**

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

return initialValue \* Math.pow(1 + growthRate, years);

}

Time & space complexity of this method = O(1)