**WEEK – 1**

**Hands-On Tool used**: Eclipse IDE

**Data Structures and Algorithms**

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

**Understand Asymptotic Notation**

**What is Big O Notation?**

Big O notation expresses how the runtime of an algorithm grows relative to the size of the input

* O(1) – Constant time (very fast)
* O(n) – Linear time (slower with larger inputs)
* O(log n) – Logarithmic time (very efficient for sorted data)
* O(n²) – Quadratic time (slow with large inputs)

|  |  |  |  |
| --- | --- | --- | --- |
| **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:**

**Product.java**;

package com.ecommerce;

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;

}

public String toString() {

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

}

}

**SearchEngine.java**

package com.ecommerce;

import java.util.Arrays;

import java.util.Comparator;

public class SearchEngine {

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) {

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

}

**SearchTest.java**

package com.ecommerce;

public class SearchTest {

public static void main(String[] args) {

Product[] products = {

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

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

new Product(103, "Sneakers", "Footwear"),

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

new Product(105, "Notebook", "Stationery")

};

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

Product result1 = SearchEngine.linearSearch(products, "Sneakers");

System.out.println(result1 != null ? result1 : "Product not found");

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

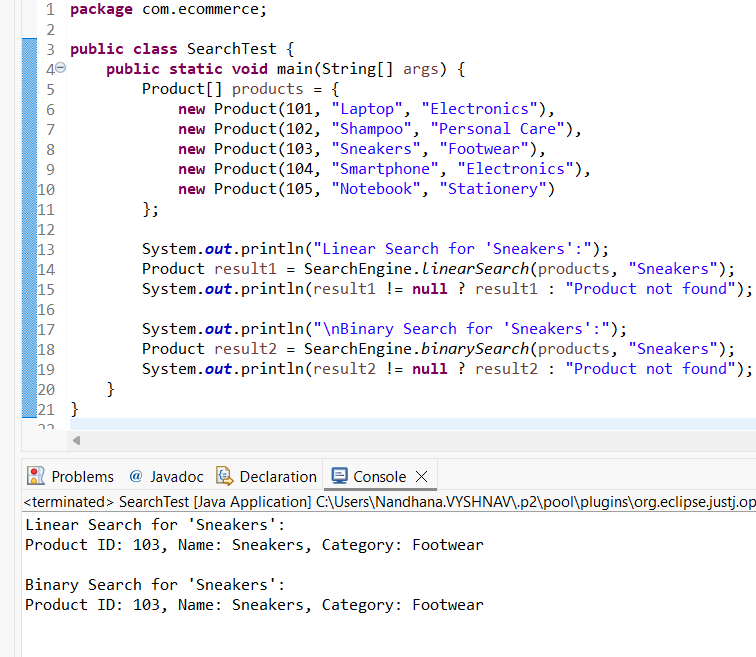
Product result2 = SearchEngine.binarySearch(products, "Sneakers");

System.out.println(result2 != null ? result2 : "Product not found");

}

}

**Output:**



**Analysis:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Sorting Needed** | **Best For** |
| Linear search | O(n) | No | Small, unsorted datasets |
| Binary search | O(log n) | Yes | Large, sorted datasets |

**Binary search** is efficient (O(log n)) but suitable only if the product list is sorted. In real-world e-commerce platforms where data is large and frequently updated, it's not practical. Instead, using HashMaps, Tries, or integrating Elasticsearch or Solr provides faster, scalable and flexible search capabilities suitable for production environments.

**Exercise 7: Financial Forecasting**

**Understand Recursive Algorithms**

**Recursion** is a programming technique where a method calls itself to solve smaller instances of the same problem.

Why use recursion?

* It simplifies repetitive calculations.
* It breaks down complex problems into base + repeated subproblem form.

**Code:**

**ForecastCalculator.java**

package com.forecasting;

public class ForecastCalculator {

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

if (years == 0) {

return initialAmount;

}

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

}

public static void main(String[] args) {

double initialAmount = 10000;

double growthRate = 0.08;

int years = 5;

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

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

}

}

**Analysis:**

The recursive algorithm has a time complexity of **O(n),** where n is the number of years. While recursion simplifies the logic, it can lead to increased memory usage and potential stack overflow for large input sizes. To optimize, the recursive approach can be replaced with an **iterative solution** or enhanced using **memoization**, both of which reduce overhead and improve performance.

**Memoized Version of Code:**

package com.forecasting;

import java.util.HashMap;

public class ForecastCalculator {

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

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

if (years == 0) {

return initialAmount;

}

if (cache.containsKey(years)) {

return cache.get(years);

}

double result = calculateFutureValue(initialAmount, growthRate, years - 1) \* (1 + growthRate);

cache.put(years, result);

return result;

}

public static void main(String[] args) {

double initialAmount = 10000;

double growthRate = 0.08;

int years = 10;

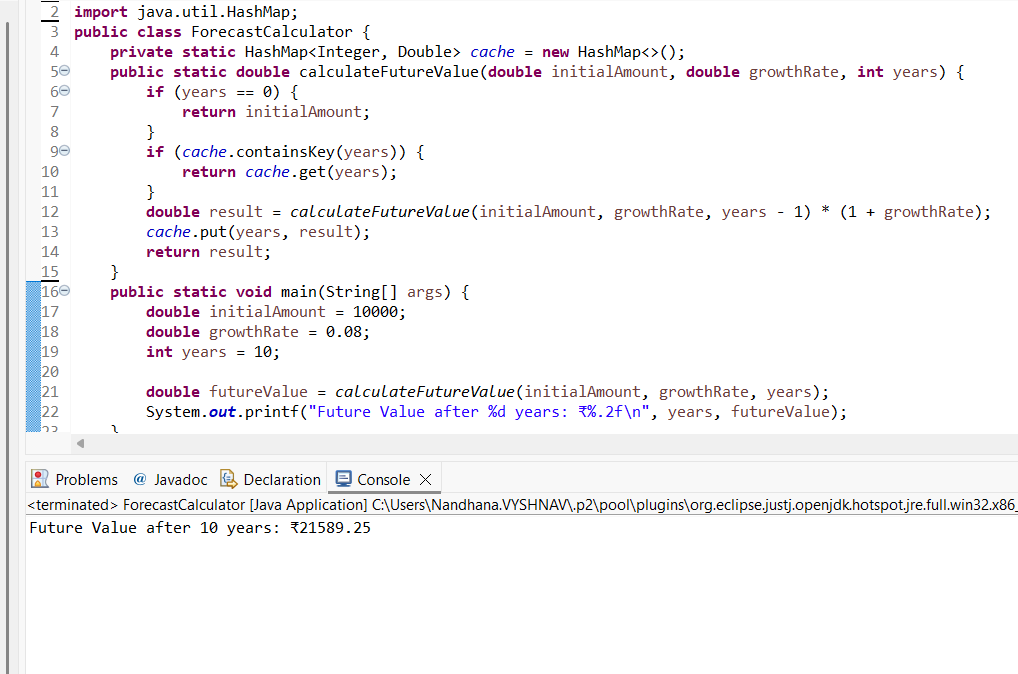
double futureValue = calculateFutureValue(initialAmount, growthRate, years);

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

}

}

**Output:**



**Conclusion:**

Recursion simplifies financial forecasting by modeling repeated growth. The recursive solution has O(n) time complexity but may cause overhead or stack overflow for large n. Optimizing via iteration or memoization improves performance and prevents excessive computation.