**WEEK – 1(Hands-on)**

**Data structures and Algorithms**

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

**Explain Big O notation and how it helps in analyzing algorithms.**

* Big O Notation describes the time or space complexity of an algorithm in terms of input size n.
* It helps understand how an algorithm scales and performs as the input grows.

**Describe the best, average, and worst-case scenarios for search operations.**

| **Case** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| Best | O(1) (first element) | O(1) (middle element) |
| Average | O(n/2) → O(n) | O(log n) |
| Worst | O(n) | O(log n) |

**Code:**

package com.example.search;

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 productId + " - " + productName + " (" + category + ")";

}

}

public class SearchUtility {

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;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

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

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

else right = mid - 1;

}

return null;

}}

import java.util.Arrays;

import java.util.Comparator;

public class Main {

public static void main(String[] args) {

Product[] products = {

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

new Product(2, "Shoes", "Footwear"),

new Product(3, "Phone", "Electronics"),

new Product(4, "Book", "Education"),

new Product(5, "Pen", "Stationery")

};

Product result1 = SearchUtility.linearSearch(products, "Phone");

System.out.println("Linear Search Result: " + result1);

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

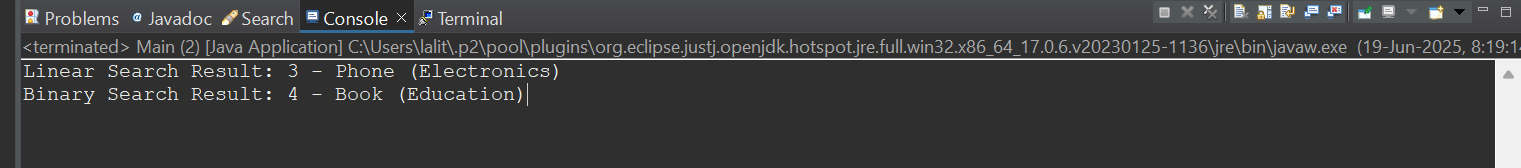
Product result2 = SearchUtility.binarySearch(products, "Book");

System.out.println("Binary Search Result: " + result2);

}

}

**OUTPUT:**

****

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

| **Algorithm** | **Time Complexity** | **Suitable For** |
| --- | --- | --- |
| Linear Search | O(n) | Small or unsorted lists |
| Binary Search | O(log n) | Large and sorted lists |

**Discuss which algorithm is more suitable for your platform and why.**

**Binary Search** is more suitable for an e-commerce platform if the product list is sorted (e.g., by name or ID), because:

* Platforms handle large volumes of product data
* Users expect fast search results
* Sorting can be done once and reused

**Exercise 7: Financial Forecasting**

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

Recursion is a technique where a method calls itself to solve smaller subproblems. It simplifies problems that have a repetitive structure (e.g., computing future values from repeated growth).

Example:  
To calculate future value after n years:

FutureValue(years) = FutureValue(years - 1) \* (1 + growthRate)

**Code:**

package com.example.forecasting;

public class FinancialForecast {

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

if (years == 0) {

return currentValue;

}

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

}

public static double forecastMemo(double currentValue, double growthRate, int years, Double[] memo) {

if (years == 0) return currentValue;

if (memo[years] != null) return memo[years];

memo[years] = *forecastMemo*(currentValue, growthRate, years - 1, memo) \* (1 + growthRate);

return memo[years];

}

}

public class Main {

public static void main(String[] args) {

double initialValue = 10000;

double growthRate = 0.08;

int years = 5;

double futureValue = FinancialForecast.*forecast*(initialValue, growthRate, years);

System.*out*.println("Recursive Future Value (5 years): " + futureValue);

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

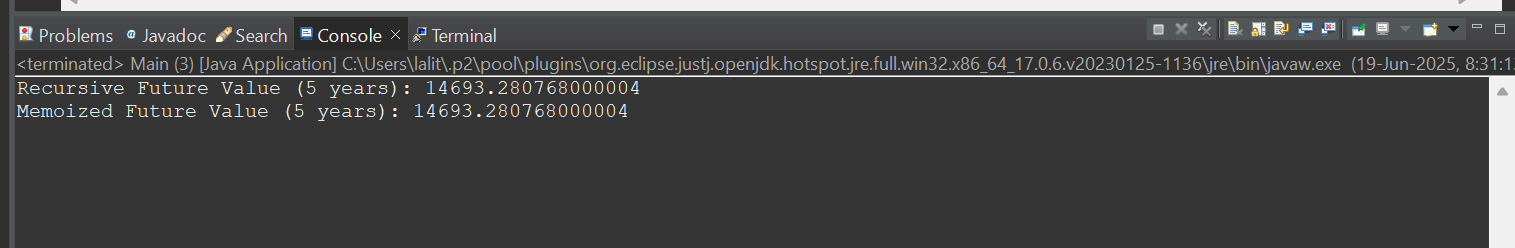
double optimizedValue = FinancialForecast.*forecastMemo*(initialValue, growthRate, years, memo);

System.*out*.println("Memoized Future Value (5 years): " + optimizedValue);

}

}

**OUTPUT:**

****

**Discuss the time complexity of your recursive algorithm.**

| **Version Time Complexity** |
| --- |
| O(n) O(n) |
| With Memoization O(n) (but avoids duplicate cells) |
| **Explain how to optimize the recursive solution to avoid excessive computation.**  **Optimized Version Using Memoization:**  public static double forecastMemo(double currentValue, double growthRate, int years, Double[] memo) {  if (years == 0) return currentValue;  if (memo[years] != null) return memo[years];  memo[years] = forecastMemo(currentValue, growthRate, years - 1, memo) \* (1 + growthRate);  return memo[years];  }  To optimize the recursive solution and avoid excessive computation, use **memoization** to cache and reuse previously calculated values. This significantly reduces function calls and makes the recursion efficient even for large inputs. |  |