**ALGORITHMS\_DATA STRUCTURES:**

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

1. **Understanding Asymptotic Notation:**

**Big O –** It describes the efficiency of an algorithm in terms of time or space complexity as input size **n** grows. Example: O(n)

**Best, Average** and **Worst case** scenarios for search operations –

* + **Best Case Scenario:**

The search operation finds the desired element at the first position or immediately. **Example**: A user searches for a product and gets the desired results in the first place only.

Time Complexity: O(1)

* + **Average Case Scenario:**

The search operation finds the desired element at normal time or somewhere in the middle. **Example**: A user searches for “Black Shoes” and found the result at the 5th position or in the middle.

Time Complexity: O(n/2)

* + **Worst Case Scenario**:

The search operation finds the desired element at the last position or not at all. **Example**: A user searches for a rare item, and it is found at the last position or not available.

Time Complexity: O(n)

1. **Code Implementation:**

* **Product.java –**

**package** com.EcommercePlatformSearch;

**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** productId + " - " + productName + " [" + category + "]";

}

}

* **LinearSearch.java –**

package com.EcommercePlatformSearch;

public class LinearSearch {

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

for (Product p : products) {

if (p.productName.equalsIgnoreCase(name)) {

return p;

}

}

return null;

}

}

* **BinarySearch.java –**

package com.EcommercePlatformSearch;

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

public static Product search(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 sortProducts(Product[] products) {

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

}

}

* **EcommerceSearchTest.java –**

package com.EcommercePlatformSearch;

public class EcommerceSearchTest {

public static void main(String[] args) {

Product[] products = {

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

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

new Product(103, "Book", "Education"),

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

};

// Linear Search

Product result1 = LinearSearch.*search*(products, "Book");

System.*out*.println("Linear Search Result: " + (result1 != null ? result1 : "Not Found"));

// Binary Search

BinarySearch.*sortProducts*(products);

Product result2 = BinarySearch.*search*(products, "Book");

System.*out*.println("Binary Search Result: " + (result2 != null ? result2 : "Not Found"));

}

}

Output:

A screenshot of a computer

AI-generated content may be incorrect.

1. **Comparing the Time Complexity of Linear and Binary search algorithms-**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Linear Search** | **Binary Search** |
| **Definition** | Scan each element one by one | Divides the searching in half |
| **Best Case** | O(1) | O(1) |
| **Average Case** | O(n) | O(log n) |
| **Worst Case** | O(n) | O(log n) |

1. **Discussing which algorithm is more suitable for our platform and why?**

For an E-Commerce Platform –

* If our product list is small or changes frequently(e.g.: live product updates)

Then **Linear Search** is simpler and more flexible.

* If our product list is very large and rarely changes

Then **Binary Search** is better as it provides fast search time.

**Exercise 7: Financial Forecasting**

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

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

It simplifies certain problems in the following steps:

* + - Breaks big tasks into smaller tasks.
    - Each term is defined using earlier ones.
    - Explore paths recursively and backtracks if needed.

1. **Code Implementation:**
   * **FinancialForecasting.java-**

**package** com.Forecasting;

**public** **class** FinancialForecasting {

**public** **static** **double** forecastValue(**double** currentValue, **double** growthRate, **int** years) {

**if** (years == 0) {

**return** currentValue;

}

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

}

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

**double** initialValue = 1000.0;

**double** annualGrowthRate = 0.10;

**int** forecastYears = 5;

**double** futureValue = *forecastValue*(initialValue, annualGrowthRate, forecastYears);

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

}

}

Output:

A screenshot of a computer

AI-generated content may be incorrect.

1. **Discuss the time complexity of our recursive algorithm.**
   * Every recursive call reduces years by 1.
   * At each level, the function performs one multiplication and one recursive call.
   * Time Complexity: O(n), where n = number of years being forecasted.
2. **Explain how to optimize the recursive solution to avoid excessive computation.**
   * Use Iteration instead of Recursion.

Recursion is fine, but for large numbers, it may cause overflow.

* + Use Memoization

Avoid recalculations

* + Use Base case

Prevents infinite recursion