**Exercise 1: E-commerce Platform Search Function**

**1.Understanding Asymptotic Notation :**

**Big O Notation**

Big O notation expresses the time or space complexity of an algorithm in the worst-case scenario, based on input size (n). It helps developers:

* Predict algorithm efficiency
* Choose the best approach for large data sets
* Identify bottlenecks in performance

**Best, Average, and Worst Case in Search**

|  |  |  |
| --- | --- | --- |
| **Case** | **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) – Last or not found | O(log n) – Not found |

**2. Setup :**

**Product.java**

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;

}

**3. Implementation :**

import java.util.Arrays;

import java.util.Comparator;

public class EcommerceSearch {

//LINEAR SEARCH IMPLEMENTATION

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

for (Product product : products) {

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

return product;

} }

return null;

}

//BINARY SEARCH IMPLEMENTATION

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (cmp == 0) {

return products[mid];

} else if (cmp < 0) {

low = mid + 1;

} else {

high = mid - 1;

} }

return null;

}

public static void main(String[] args) {

Product[] products = {

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

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

new Product(103, "Shoes", "Fashion"),

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

};

String target = "Phone";

Product result1 = linearSearch(products, target);

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

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

Product result2 = binarySearch(products, target);

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

}

}

**4. Analysis :**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Linear Search** | **Binary Search** |
| **Time Complexity** | O(n) | O(log n) |
| **Requires Sorting** | No | Yes |
| **Suitable for** | Small/Unsorted | Large/Sorted Arrays |

**Which is Better?**

* **Binary Search** is faster **(O(log n)),** making it ideal for **large, sorted datasets.**
* **Linear Search** isuseful for **unsorted** or **small datasets** due to its simplicity.

**Exercise 2: Financial Forecasting**

**1.Understanding Recursive Algorithms :**

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

**Use of Recursion :**

* Simplifies problems that can be broken into sub-problems (like computing compound values over years).
* Commonly used in problems like factorials, Fibonacci series, and financial predictions

**2. Setup and Implementation :**

public class FinancialForecasting {

public static double futureValueRecursive(double presentValue, double rate, int years) {

if (years == 0) {

return presentValue;

} return futureValueRecursive(presentValue, rate, years - 1) \* (1 + rate);

}

public static void main(String[] args) {

double presentValue = 10000;

double growthRate = 0.08;

int years = 5;

double futureValue = futureValueRecursive(presentValue, growthRate, years);

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

}

}

**4. Analysis :**

**Time Complexity**

* Each recursive call reduces years by 1.
* So, total calls = n
* **Time Complexity:** O(n)
* **Space Complexity:** O(n) due to the call stack

For large n, recursion can be inefficient due to:

* Stack overflow risk
* Repeated calculations (in more complex recursive scenarios)