**WEEK-1 Algorithms Data Structures HandsOn**

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

**Product.java**

package module2;

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;

}

}

**ProductIdComparator.java**

package module2;

import java.util.Comparator;

public class ProductIdComparator implements Comparator<Product> {

public int compare(Product p1, Product p2) {

return p1.productId - p2.productId;

}

}

**ProductSearch.java**

package module2;

import java.util.Arrays;

import java.util.Scanner;

public class ProductSearch {

public static Product linearSearch(Product[] products, int targetId) {

for (Product p : products) {

if (p.productId == targetId) {

return p;

}

}

return null;

}

// Binary Search

public static Product binarySearch(Product[] products, int targetId) {

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

if (products[mid].productId == targetId) {

return products[mid];

} else if (products[mid].productId < targetId) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

Product[] products = {

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

new Product(1, "Jeans", "Fashion"),

new Product(5, "Phone", "Electronics"),

new Product(2, "Rings", "Fashion"),

new Product(4, "Shirt", "Fashion")

};

Arrays.sort(products, new ProductIdComparator());

System.out.print("Enter Product ID to search: ");

int searchId = sc.nextInt();

System.out.println("\n=== Linear Search ===");

Product foundLinear = linearSearch(products, searchId);

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

System.out.println("\n=== Binary Search ===");

Product foundBinary = binarySearch(products, searchId);

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

sc.close();

}

}

**ANALYSIS**

**Linear Search** has a **best-case time complexity of O(1)**, which occurs when the target element is found at the very beginning of the array. The **average case** is approximately **O(n/2)**, which simplifies to **O(n)**, assuming the element is found somewhere in the middle. The **worst-case time complexity is O(n)**, occurring when the element is at the end or not present at all. Therefore, the overall time complexity of Linear Search is **O(n)**.

**Binary Search**, on the other hand, is more efficient but requires the array to be **sorted**. In the **best case**, the target element is found at the middle of the array, resulting in a time complexity of **O(1)**. Both the **average** and **worst-case complexities** are **O(log n)**, as the algorithm repeatedly divides the search interval in half. Hence, the overall time complexity of Binary Search is **O(log n)**.

In summary, **Linear Search** is simpler and works on unsorted data but is less efficient for large inputs. **Binary Search** offers much better performance but requires a sorted array.

**Since e-commerce platforms have large and mostly sorted product catalogs, Binary Search is the better choice for performance and scalability.**

**OUTPUT:**

**A computer screen shot of a program

AI-generated content may be incorrect.**

**Exercise 7: Financial Forecasting**

**FinancialForecast.java**

package module2;

import java.util.Scanner;

public class FinanciaForecast {

public static double futureValue(double principal, double rate, int years) {

if (years == 0) {

return principal;

} else {

return (1 + rate) \* *futureValue*(principal, rate, years - 1);

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.*in*)

System.*out*.print("Enter Principal Amount: ");

double principal = sc.nextDouble();

System.*out*.print("Enter Annual Growth Rate (e.g., 0.08 for 8%): ");

double rate = sc.nextDouble();

System.*out*.print("Enter Number of Years: ");

int years = sc.nextInt();

double result = *futureValue*(principal, rate, years);

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

sc.close();

}

}

**ANALYSIS**

**Time Complexity**

The recursive method calls itself n times (n is the number of years). So, Time Complexity = O(n).

**Optimization:**

Recursive calls can cause stack overflow for large n.

**Use Iteration Instead of Recursion:**

public static double futureValueIterative(double principal, double rate, int years) {

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

principal \*= (1 + rate);

}

return principal;

}

**Use iteration instead of recursion because it is more efficient, uses constant memory, and avoids stack overflow in large inputs.**

**OUTPUT:**

**A screenshot of a computer program

AI-generated content may be incorrect.**