**WEEK -1**

**Data Structures and Algorithms**

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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Understand Asymptotic Notation**

* Big O notation is used to show how fast or slow an algorithm is, especially when the size of input increases.
* In search operations, Big O is used to compare which algorithm will perform better as data grows.

**Time Complexity:**

* Best Case: The element is found at the very beginning or at the mid position.

Linear Search: O(1)

Binary Search: O(1)

* Average Case: The element is somewhere in the middle.

Linear Search: O(n/2) = O(n)

Binary Search: O(log n)

* Worst Case: The element is at the end or if it is not present.

Linear Search: O(n)

Binary Search: O(log n)

**Code:**

import java.util.Arrays;

import java.util.Comparator;

// Product class

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 + " ";

}

}

public class Main {

// Linear Search

public static int ls(Product[] products, String key) {

for (int i = 0; i < products.length; i++) {

if (products[i].productName.equalsIgnoreCase(key)) {

return i;

}

}

return -1;

}

// Binary Search

public static int bs(Product[] products, String key) {

int l = 0;

int r = products.length - 1;

while (l<=r) {

int mid = (l+r)/2;

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

if (cmp == 0) return mid;

else if (cmp < 0) l = mid + 1;

else r = mid - 1;

}

return -1;

}

public static void main(String[] args) {

Product[] products = {

new Product(101, "Biryani", "Food"),

new Product(102, "Pant", "Clothing"),

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

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

new Product(105, "Cake", "Bakery")

};

// Linear Search

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

int result1= ls(products, "Cake");

System.out.println(result1 != -1 ? "Product Found: " + products[result1] : "Product not found");

// Sorting the array before implementing binary search

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

// Binary Search

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

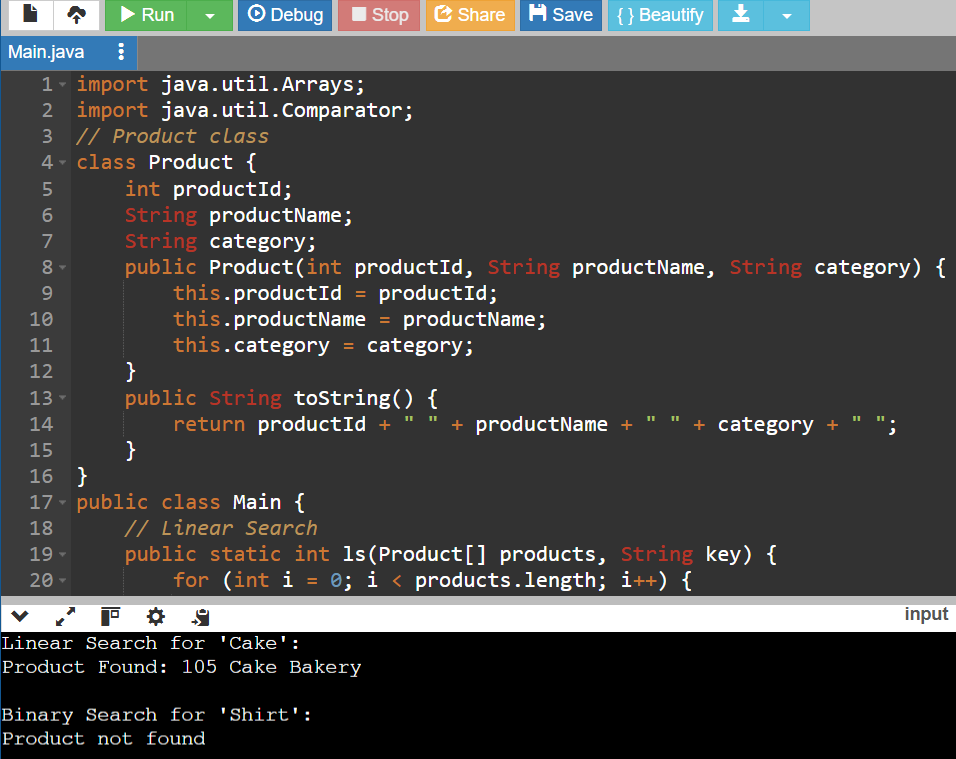
int result2 = bs(products, "Shirt");

System.out.println(result2 != -1 ? "Product Found: " + products[result2] : "Product not found");

}

}

**Output:**



**Analysis:**

* Linear Search traverse throughout the array once, element by element, so its time complexity is O(n).
* Binary Search divides the array in half at each step (but it works only when array is sorted ), so its time complexity is O(log n).

**Which is better?**

* For small or unsorted arrays, linear search is preferred.
* For large data or when the data is sorted, binary search is faster and more efficient.
* As e-commerce platforms usually deal with huge amounts of products, binary search is more suitable when the data can be kept sorted.

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Concept of recursion:**

Recursion is a technique where a method calls itself to solve smaller parts of a bigger problem.  
Examples: factorial, Fibonacci series, and tree traversal.

**Code:**

import java.util.\*;

public class Main {

// Recursion to calculate future value

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

if (years == 0) {

return currentValue;

}

// Applying recursively

return forecast(currentValue \* (1 + rate), rate, years – 1);

}

public static void main(String[] args) {

Scanner sc=new Scanner(System.in);

System.out.printf(“Enter Principal amount while taking loan: “);

double presentValue = sc.nextDouble();

System.out.printf(“Enter Rate Of Interest: “);

double rate = sc.nextDouble()/100;

System.out.printf(“Enter No of Years: “);

int years = sc.nextInt();

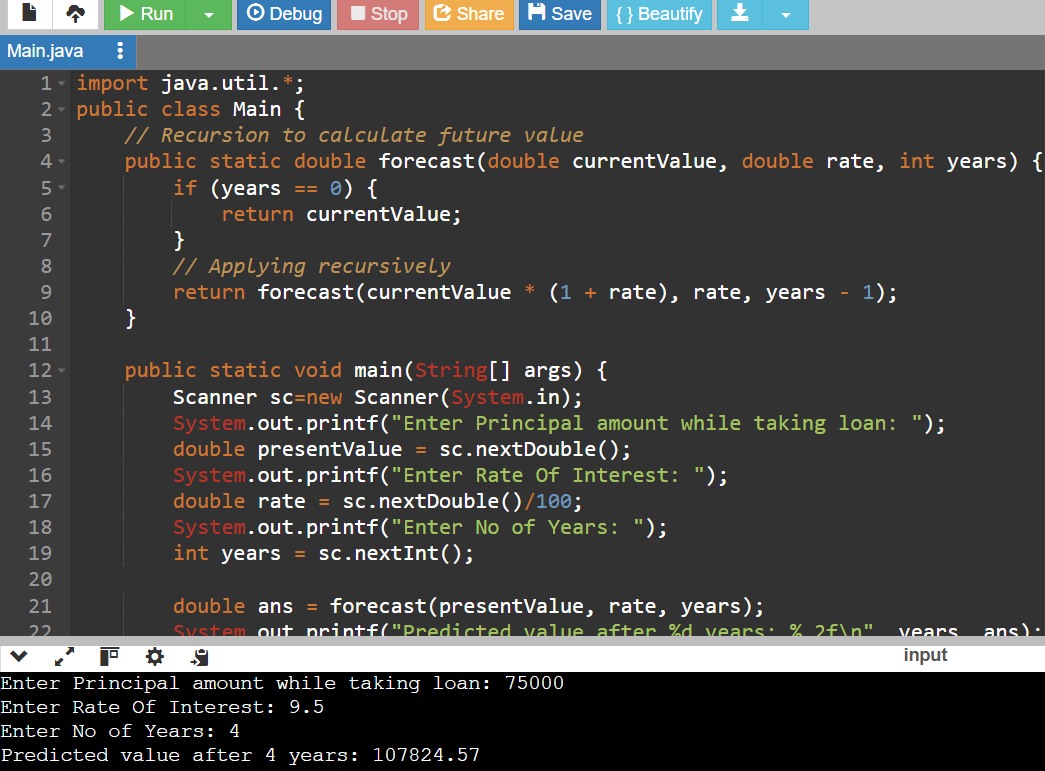
double ans = forecast(presentValue, rate, years);

System.out.printf(“Predicted value after %d years: %.2f\n”, years, ans);

}

}

**Output:**



**Explanation:**

* I used a recursive function to forecast the financial value by applying a fixed growth rate each year.
* The function keeps multiplying the current value with (1 + rate) until the number of years becomes zero.
* It’s a simple and clean approach when we only need the final result.
* If I had to calculate values for many years or intervals repeatedly, using a loop or storing results would be more efficient.

**Analysis:**

Complexity of Recursive Algorithm:

* The recursive method calls itself once for every year.
* Therefore, the time complexity is O(n), where n is the number of years.
* The space complexity is also O(n), because each recursive call adds a new frame to the call stack.

How to Optimize the Recursive Solution:

It can be optimised using iterative approaches and memorization.