**Data Structures and Algorithms**

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

**1)**

a. Big O notation helps us analyze how efficient an algorithm is as the input grows. It shows the worst-case or average-case time taken by an algorithm.

#### b. **Best, Average, and Worst Case**

* **Best case**: Fastest time (e.g., item is at the beginning).
* **Average case**: Expected position (e.g., item is in the middle).
* **Worst case**: Longest time (e.g., item is at the end or not found).

**2) CODE:**

class Product {

int id;

String name;

String category;

Product(int id, String name, String category) {

this.id = id;

this.name = name;

this.category = category;

}

}

public class Main {

public static Product linearSearch(Product[] items, String searchName) {

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

if (items[i].name.equals(searchName)) {

return items[i];

}

}

return null;

}

public static Product binarySearch(Product[] items, String searchName) {

int start = 0;

int end = items.length - 1;

while (start <= end) {

int mid = (start + end) / 2;

int result = items[mid].name.compareTo(searchName);

if (result == 0) {

return items[mid];

} else if (result < 0) {

start = mid + 1;

} else {

end = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Product[] list = new Product[3];

list[0] = new Product(1, "Shirt", "Clothing");

list[1] = new Product(2, "Laptop", "Electronics");

list[2] = new Product(3, "Book", "Stationery");

long start1 = System.nanoTime();

Product result1 = linearSearch(list, "Laptop");

long end1 = System.nanoTime();

if (result1 != null) {

System.out.println("Found using linear search: " + result1.name + " in " + result1.category);

} else {

System.out.println("Not found using linear search.");

}

System.out.println("Linear search time (ns): " + (end1 - start1));

java.util.Arrays.sort(list, (a, b) -> a.name.compareTo(b.name));

long start2 = System.nanoTime();

Product result2 = binarySearch(list, "Laptop");

long end2 = System.nanoTime();

if (result2 != null) {

System.out.println("Found using binary search: " + result2.name + " in " + result2.category);

} else {

System.out.println("Not found using binary search.");

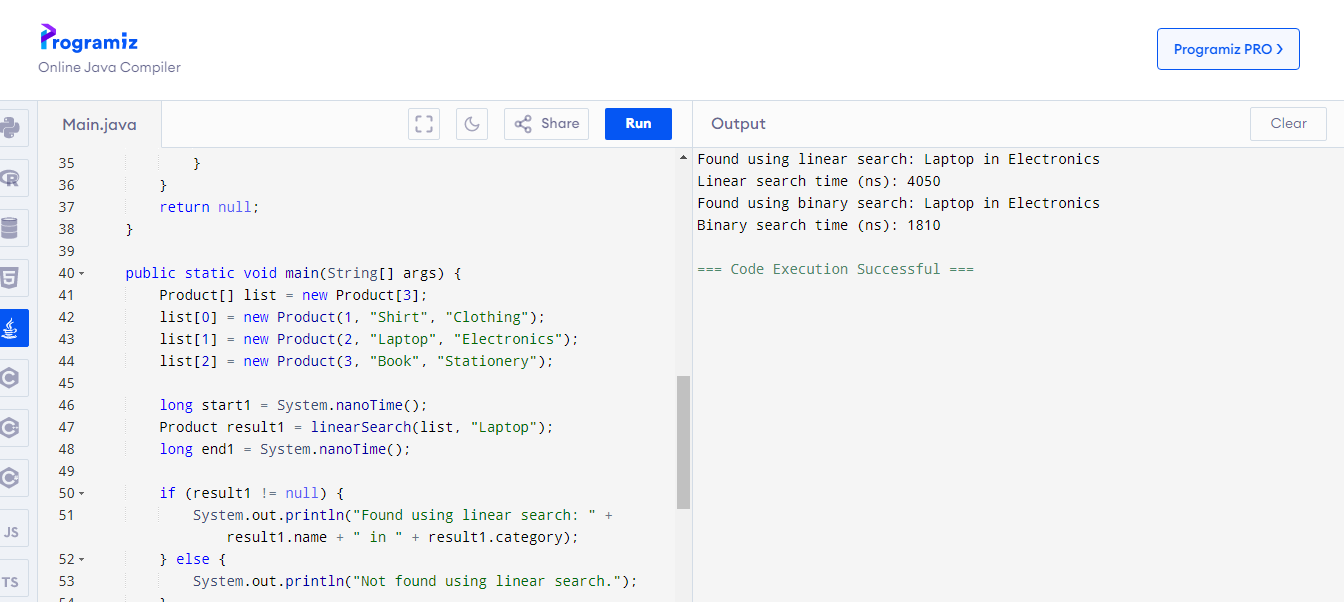
}

System.out.println("Binary search time (ns): " + (end2 - start2));

}

}

**OUTPUT:**



**Linear search** takes **O(n)** time and is used when data is **unsorted.**  
**Binary search** takes **O(log n)** time and is better when data is **sorted** for faster searching.

**Exercise 7: Financial Forecasting**

Recursion is a programming technique where a function calls itself to solve a smaller part of a larger problem. It simplifies complex problems by breaking them down into smaller, repeatable tasks.

**CODE**:

import java.util.Scanner;

public class Main {

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

if (years == 0) {

return amount;

}

return forecast(amount \* (1 + rate / 100), rate, years - 1);

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("FINANCIAL FORECASTING TOOL\n");

System.out.print("Enter initial amount: ");

double amount = sc.nextDouble();

System.out.print("Enter annual growth rate (%): ");

double rate = sc.nextDouble();

System.out.print("Enter number of years: ");

int years = sc.nextInt();

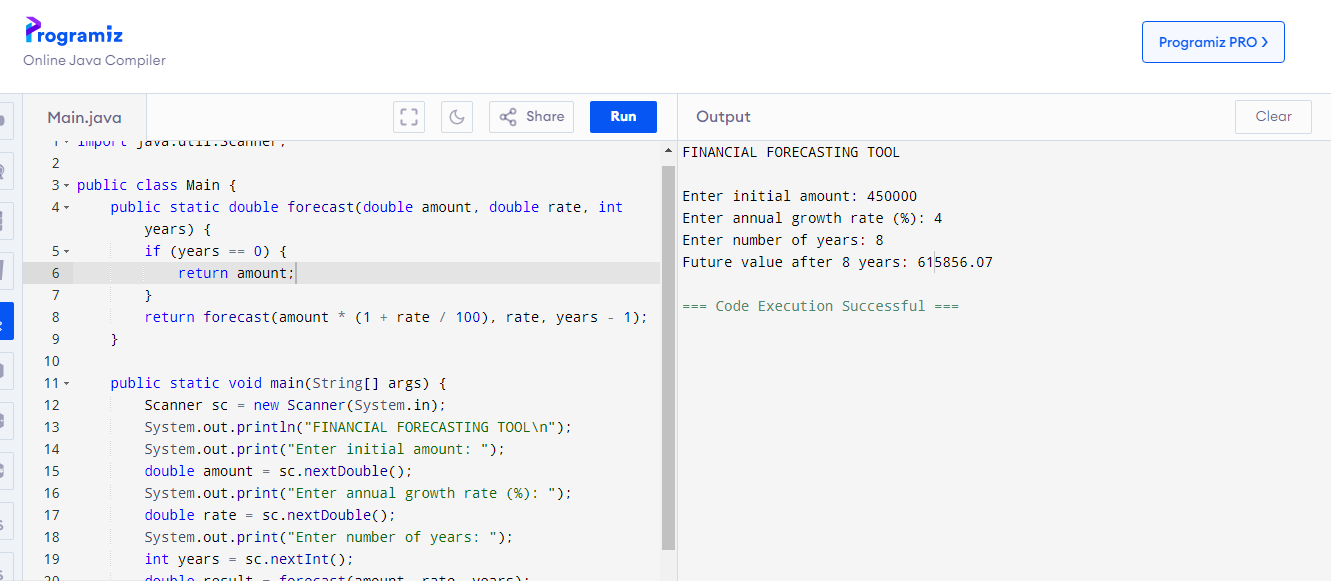
double result = forecast(amount, rate, years);

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

}

}

**OUTPUT:**



**Time Complexity:**  
The recursive algorithm makes one call per year, so the time complexity is **O(n)** where n is the number of years.

**Optimization:**

For this specific problem, recursion is already efficient since there are no repeated subproblems.

If future calculations had overlapping subproblems, we could use **memoization** or **convert to iteration** to optimize further.

*Ankita S Patil*

*Email: ankitaaapatil21@gmail.com*

*Superset id: 6390359*