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

Big O notation is used to describe the time complexity and space complexity of algorithms. It helps us analyze how an algorithm's performance scales with input size

* To compare different algorithms independently of hardware.
* To identify potential bottlenecks in large input sizes.
* To help choose the most efficient algorithm for a specific use case.

| **Scenario** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Best Case** | O(1) (first match) | O(1) (middle match) |
| **Average Case** | O(n/2) ≈ O(n) | O(log n) |
| **Worst Case** | O(n) (last or not found) | O(log n) |

***Product.java***

package e2\_E\_commercePlatformSearchFunction;

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 e2\_E\_commercePlatformSearchFunction;

public class LinearSearch {

public static Product search(Product[] products, int id) {

for (Product p : products) {

if (p.productId == id) {

return p;

}

}

return null;

}

}

***BinarySearch.java***

package e2\_E\_commercePlatformSearchFunction;

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

public static Product search(Product[] products, int id) {

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

while (low <= high) {

int mid = (low + high) / 2;

if (products[mid].productId == id)

return products[mid];

else if (products[mid].productId < id)

low = mid + 1;

else

high = mid - 1;

}

return null;

}

public static void sortProducts(Product[] products) {

Arrays.*sort*(products, Comparator.*comparingInt*(p -> p.productId));

}

}

***Main.java***

package e2\_E\_commercePlatformSearchFunction;

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(103, "Laptop", "Electronics"),

new Product(101, "Shoes", "Fashion"),

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

new Product(104, "Book", "Education")

};

Product foundLinear = LinearSearch.*search*(products, 104);

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

BinarySearch.*sortProducts*(products);

Product foundBinary = BinarySearch.*search*(products, 104);

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

BinarySearch.*sortProducts*(products);

Product NotfoundBinary = BinarySearch.*search*(products, 105);

System.***out***.println("Binary Search Result: " + (NotfoundBinary != null ? foundBinary : "Not found"));

}

}

***Output:***

A computer screen shot of a program

AI-generated content may be incorrect.

***Time Complexity:***

| **Algorithm** | **Time Complexity** |
| --- | --- |
| Linear Search | O(n) |
| Binary Search | O(log n) |

**Exercise 7: Financial Forecasting**

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

It consists of:

* Base case: The simplest case that ends recursion.
* Recursive case: The function calls itself with a simpler input.
* It simplifies problems involving repetitive or hierarchical structures.
* Useful for calculations like Fibonacci series, factorials, and compound interest growth.

**Future Value = Present Value × (1 + growthRate)^years**

***FinancialForecast.java***

package e7\_FinanciaForecasting;

import java.util.Scanner;

public class FinancialForecast {

public static double forecastValue(double presentValue, double growthRate, int years) {

if (years == 0) {

return presentValue;

}

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

}

public static void main(String[] args) {

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

System.***out***.print("Enter the present value (initial amount): ");

double presentValue = sc.nextDouble();

System.***out***.print("Enter the annual growth rate (e.g., enter 5% as 0.05): ");

double growthRate = sc.nextDouble();

System.***out***.print("Enter the number of years to forecast: ");

int years = sc.nextInt();

double futureValue = *forecastValue*(presentValue, growthRate, years);

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

}

}

***Output:***

A computer screen shot of a program

AI-generated content may be incorrect.

**Time Complexity:**

The function is called once per year, so:

Time Complexity = O(n**)**, where n = number of years

Space Complexity = O(n) due to call stack usage

**How to Optimize Recursive Solution:**

* Tail Recursion: Some languages optimize tail calls, but Java does not reliably do tail call optimization.
* Memorization: Cache already computed results to avoid redundant calculations.
* Convert to Iteration: Iterative solutions are more memory efficient.