**Mandatory HandsOn**

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

**Big O Notation (Asymptotic Notation)**

**Big O Notation** is a mathematical way to describe how fast or slow an algorithm runs as the size of input increases. It helps in predicting the efficiency of code without running it. It tells us how much time or space an algorithm will take in different conditions.

**Search Operation Cases**

**Best Case:**

The product is found immediately (first try).

Time complexity:

* Linear Search: O(1)
* Binary Search: O(1)

**Average Case:**

The product is somewhere in the middle.

Time complexity:

* Linear Search: O(n)
* Binary Search: O(log n)

**Worst Case:**

The product is not found or is the last in the list.

Time complexity:

* Linear Search: O(n)
* Binary Search: O(log n)

**Program:**

import java.util.Arrays;

class Product {

int productId;

String productName;

String category;

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

this.productId = id;

this.productName = name;

this.category = category;

}

public String toString() {

return productId + " | " + productName + " | " + category;

}

}

public class EcommerceSearch {

static int linearSearch(Product[] products, String target) {

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

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

return i;

}

}

return -1;

}

static int binarySearch(Product[] products, String target) {

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

while (low <= high) {

int mid = (low + high) / 2;

int result = products[mid].productName.compareToIgnoreCase(target);

if (result == 0)

return mid;

else if (result < 0)

low = mid + 1;

else

high = mid - 1;

}

return -1;

}

public static void main(String[] args) {

Product[] products = {

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

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

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

new Product(104, "Shoes", "Footwear"),

new Product(105, "Watch", "Accessories")

};

String searchItem = "Watch";

int foundIndex = linearSearch(products, searchItem);

if (foundIndex != -1)

System.out.println("Linear Search Result: " + products[foundIndex]);

else

System.out.println("Linear Search: Product not found");

Arrays.sort(products, (a, b) -> a.productName.compareToIgnoreCase(b.productName));

int binaryIndex = binarySearch(products, searchItem);

if (binaryIndex != -1)

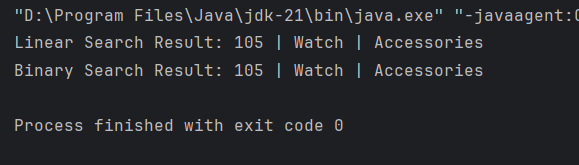
System.out.println("Binary Search Result: " + products[binaryIndex]);

else

System.out.println("Binary Search: Product not found");

}

}



**Analysis:**

**Linear Search:**

* Simple, does not require sorting.
* Slower on large datasets.
* Time Complexity: **O(n)**

**Binary Search:**

* Much faster on large datasets.
* Requires sorted data before searching.
* Time Complexity: **O(log n)**

**Exercise 7: Financial Forecasting**

**Recursion**

Recursion is when a method calls itself to solve smaller parts of the same problem. Think of it like peeling an onion layer by layer until you reach the core. Once the smallest part is solved, the results build back up to solve the full problem. Recursion helps break down repeated patterns, like calculating compound interest year after year, into smaller and simpler steps.

**Program:**

public class FinancialForecast {

static double forecastRecursive(double currentAmount, double rate, int years) {

if (years == 0) {

return currentAmount;

}

return forecastRecursive(currentAmount, rate, years - 1) \* (1 + rate);

}

static double forecastIterative(double currentAmount, double rate, int years) {

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

currentAmount \*= (1 + rate);

}

return currentAmount;

}

public static void main(String[] args) {

double presentValue = 15000;

double growthRate = 0.07;

int years = 6;

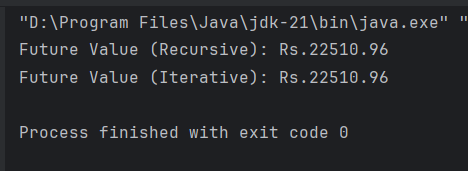
double recursiveResult = forecastRecursive(presentValue, growthRate, years);

double iterativeResult = forecastIterative(presentValue, growthRate, years);

System.out.printf("Future Value (Recursive): Rs.%.2f\n", recursiveResult);

System.out.printf("Future Value (Iterative): Rs.%.2f\n", iterativeResult);

}

}

**Analysis:**

Recursive Approach: O(n) → One call per year (n years)

Iterative Approach: O(n) → One loop iteration per year