**Data Structures and Algorithms**

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

Big O notation is a powerful tool used in computer science to describe the time complexity or space complexity of algorithms. Big-O is a way to express the upper bound of an algorithm’s time or space complexity.

How Big O helps in analyzing algorithms:

**1. Comparing Algorithms:**

Big O notation allows for a standardized comparison of different algorithms' efficiency, helping in selecting the best one for a specific task or dataset.

**2. Predicting Scalability:**

By understanding how an algorithm's runtime or memory usage scales, developers can predict how it will perform with larger inputs and make informed decisions about its suitability for different scenarios.

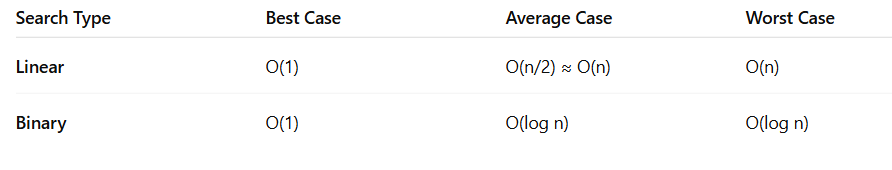
**3. Optimizing Performance:**

Big O helps identify potential bottlenecks in algorithms, enabling developers to optimize performance by focusing on the most critical parts of the code.

**4. Choosing the Right Data Structures:**

Big O notation helps in selecting appropriate data structures that offer optimal performance for specific operations.

Best, average, and worst-case scenarios for search operations:

**Product.java:**

package E\_Commerce;

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;

}

}

**LinearSearch.java:**

package E\_Commerce;

public class LinearSearch {

public static Product linearSearch(Product[] products, String targetName) {

for (Product p : products) {

if (p.productName.equalsIgnoreCase(targetName)) {

return p;

}

}

return null;

}

}

**BinarySearch.java:**

package E\_Commerce;

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch {

public static Product binarySearch(Product[] products, String targetName) {

int left = 0, right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0) return products[mid];

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

else right = mid - 1;

}

return null;

}

}

**Main.java:**

package E\_Commerce;

import java.util.Arrays;

import java.util.Comparator;

public class Main {

public static void main(String[] args) {

Product[] products = {

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

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

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

new Product(104, "Table", "Furniture")

};

Product found1 = LinearSearch.*linearSearch*(products, "Mobile");

System.*out*.println("Linear Search Found: " + (found1 != null ? found1 : "Not Found"));

Product found2 = LinearSearch.*linearSearch*(products, "tablet");

System.*out*.println("Linear Search Found: " + (found2 != null ? found2 : "Not Found"));

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

Product found3 = BinarySearch.*binarySearch*(products, "Mobile");

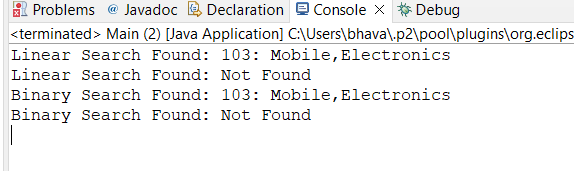
System.*out*.println("Binary Search Found: " + (found3 != null ? found3 : "Not Found"));

Product found4 = BinarySearch.*binarySearch*(products, "tablet");

System.*out*.println("Binary Search Found: " + (found4 != null ? found4 : "Not Found"));

}

}

**Output:**

**** Binary Search is more suitable for large datasets due to its logarithmic performance, but it requires the array to be sorted.

 Linear Search is simpler and works on unsorted data, useful for small datasets or dynamic collections.

Recommendation for E-commerce Platform:

* Use Binary Search for static product catalogs (with sorted data).

**Exercise 7: Financial Forecasting**

**Recursion** is when a method calls itself to solve smaller sub-problems of the original problem.

A recursive function must have:

* **Base Case**: Stops further recursion.
* **Recursive Case**: Breaks the problem into smaller parts.

Example Use Cases:

* Factorial, Fibonacci sequence, Tree traversal, etc.

**Forecasting Future Value:**

Assume:

• There is a current value.

• Company has a fixed annual growth rate.

• You want to know the value after n years.

**FinancialForecast.java:**

package FinancialForecast;

public class FinancialForecast {

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

if (years == 0) {

return presentValue;

}

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

}

public static void main(String[] args) {

double presentValue = 10000;

double annualRate = 0.08;

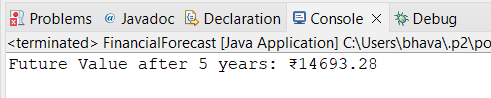
int years = 5;

double futureVal = *futureValue*(presentValue, annualRate, years);

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

}

}

**Output:**

**Time Complexity:**

• Recursive depth = years

• So Time Complexity = O(n)

**Space Complexity:**

• Due to recursion stack: O(n)

**Optimization:**

To avoid stack overflow in deep recursion (e.g., years = 10000), use iteration or tail recursion.

**Iterative Version:**

This version uses O(1) space and is more efficient for large inputs.