Algorithms Data Structures

**Exercise 1: Inventory Management System**

**Scenario:**

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

Code:

Big O notation describes the **upper bound** of an algorithm’s time or space complexity in terms of input size n. It helps estimate how an algorithm performs as data scales.

#### **Best, Average, Worst Case in Search**

* **Best Case**: The item is found at the first check.
* **Average Case**: The item is somewhere in the middle.
* **Worst Case**: The item is at the end or not present.

//Product.java

public class Product implements Comparable<Product> {

int productId;

String productName;

String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

@Override

public int compareTo(Product other) {

return this.productName.compareToIgnoreCase(other.productName);

}

@Override

public String toString() {

return "ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

//ProductSearch.java

import java.util.Arrays;

public class ProductSearch {

// Linear Search

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

for (Product p : products) {

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

return p;

}

}

return null;

}

// Binary Search

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

int comparison = products[mid].productName.compareToIgnoreCase(name);

if (comparison == 0)

return products[mid];

else if (comparison < 0)

low = mid + 1;

else

high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shoes", "Fashion"),

new Product(3, "Mobile", "Electronics"),

new Product(4, "Bag", "Fashion"),

new Product(5, "Tablet", "Electronics")

};

System.out.println("Linear Search:");

Product found1 = linearSearch(products, "Mobile");

System.out.println(found1 != null ? found1 : "Product not found");

Arrays.sort(products);

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

Product found2 = binarySearch(products, "Mobile");

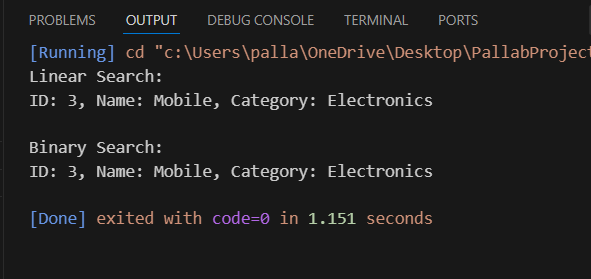
System.out.println(found2 != null ? found2 : "Product not found");

}

}

|  |  |  |
| --- | --- | --- |
| **Search Type** | **Time Complexity** | **Sorted Required?** |
| Linear Search | O(n) | ❌ No |
| Binary Search | O(log n) | ✅ Yes |

**OUTPUT:**



**Exercise 7: Financial Forecasting**

**Scenario:**

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

Code:

Recursion is a method where a function calls itself to solve a smaller version of the same problem.

It simplifies complex problems (like traversing trees, calculating factorials, etc.).

public class FinancialForecast {

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

if (years == 0) {

return presentValue;

}

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

}

public static void main(String[] args) {

double presentValue = 10000;

double growthRate = 0.08;

int years = 5;

double futureVal = futureValue(presentValue, growthRate, years);

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

}

}

Time Complexity:

Each recursive call reduces years by 1.

So, **Time Complexity = O(n)**, where n is the number of years.

**OUTPUT:**

