Data structures and Algorithms

Exercise 2: E-commerce Platform Search Function

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**Theory Portion:**

**🔹 Big O Notation**

**Big O notation** describes the upper bound of an algorithm's running time as the input size grows. It helps us evaluate algorithm performance and scalability.

**🔹 Best, Average, and Worst-Case Scenarios**

For search algorithms:

* **Linear Search**:
  + Best: O(1) (element is first)
  + Average: O(n/2) ≈ O(n)
  + Worst: O(n) (element not found or last)
* **Binary Search**:
  + Best: O(1) (element is middle)
  + Average/Worst: O(log n) (divide-and-conquer)

**🔹Analysis**

The time complexity of Linear Search is O(n) and space complexity is O(1) and sorting is not required.

The time complexity of Binary Search is O(log n) and space complexity is O(1) and sorting is required.

**Which is More Suitable?**

* **Small datasets or unsorted data**: Linear search is acceptable.
* **Large datasets and frequent searches**: Binary search is faster, but requires sorted data.

For an **e-commerce platform** with many products and frequent searches, **Binary Search** is best.

**Code:**

Product.java

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;

}

@Override

public String toString() {

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

}

}

SearchAlgorithms.java

import java.util.Arrays;

import java.util.Comparator;

public class SearchAlgorithms {

// Linear Search

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

for (Product product : products) {

if (product.productName.equalsIgnoreCase(productName)) {

return product;

}

}

return null;

}

// Binary Search (requires sorted array by productName)

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

// Sort array before binary search

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

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

while (low <= high) {

int mid = low + (high - low) / 2;

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

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

else if (cmp < 0) high = mid - 1;

else low = mid + 1;

}

return null;

}

}

Main.java

public class Main {

public static void main(String[] args) {

Product[] products = {

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

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

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

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

new Product(105, "Tablet", "Electronics")

};

String searchName = "Phone";

// Linear Search

Product resultLinear = SearchAlgorithms.linearSearch(products, searchName);

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

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

// Binary Search

Product resultBinary = SearchAlgorithms.binarySearch(products, searchName);

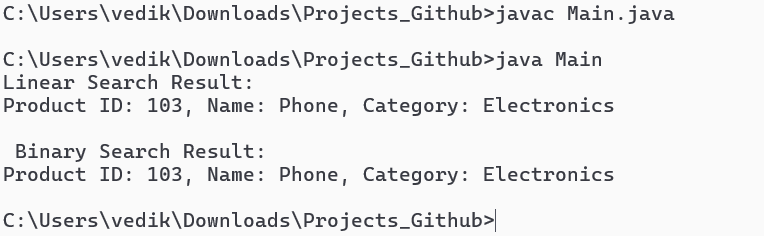
System.out.println("\n Binary Search Result:");

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

}

}

**Output:**



Exercise 7: Financial Forecasting

**Scenario:**

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

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**Theory Portion:**

**🔹 What is Recursion?**

**Recursion** is a technique where a method calls itself to solve a smaller instance of a problem. It's especially helpful for problems that can be broken down into **repetitive subproblems** like factorial, Fibonacci series, etc.

**🔹 Time Complexity**

**🔸 Recursive Method**

* **Time Complexity**: O(n)  
  → One recursive call per year, so linear with respect to n.

**🔸 With Memoization**

* **Time Complexity**: Still O(n), but avoids **repeated calculations** (beneficial in more complex recursive formulas).
* **Space Complexity**: O(n) due to call stack and memo array.

**🔹Optimization**

* **Tail Recursion**: Not much benefit here due to multiplication order.
* **Memoization**: Useful when values are reused or recursion is nested.
* **Iterative Approach**: Can replace recursion in practice to avoid stack overflow.

**Code:**

FinancialForecast.java

public class FinancialForecast {

// Recursive method to calculate future value

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

if (years == 0) {

return presentValue;

} else {

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

}

}

// Optimized version using memoization (optional)

public static double calculateFutureValueMemo(double presentValue, double rate, int years, double[] memo) {

if (years == 0) {

return presentValue;

}

if (memo[years] != 0) {

return memo[years];

}

memo[years] = (1 + rate) \* calculateFutureValueMemo(presentValue, rate, years - 1, memo);

return memo[years];

}

public static void main(String[] args) {

double presentValue = 10000; // ₹10,000

double rate = 0.08; // 8% growth per year

int years = 5;

// Plain recursion

double futureValue = calculateFutureValue(presentValue, rate, years);

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

// Using memoization

double[] memo = new double[years + 1];

double futureValueMemo = calculateFutureValueMemo(presentValue, rate, years, memo);

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

}

}

**Output:**

