**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.

**SOLUTION:**

**Big O Notation:**

Big O notation describes the upper bound of an algorithm’s running time as the input size increases.

**Time Complexities:**

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Linear Search** | **Binary Search** |
| **Best Case** | **O(1)** | **O(1)** |
| **Average Case** | **O(n)** | **O(log n)** |
| **Worst Case** | **O(n)** | **O(log n)** |

**CODE:**

**Product.java**

package ecommerce;

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 + ")";

}

}

**SearchUtility.java**

package ecommerce;

import java.util.Arrays;

import java.util.Comparator;

public class SearchUtility {

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

for (Product product : products) {

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

return product;

}

}

return null;

}

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

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

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

else high = mid - 1;

}

return null;

}

public static void sortByName(Product[] products) {

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

}

}

**TestSearch.java**

package ecommerce;

public class TestSearch {

public static void main(String[] args) {

Product[] products = {

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

new Product(2, "Shirt", "Clothing"),

new Product(3, "Book", "Education"),

new Product(4, "Headphones", "Electronics"),

new Product(5, "Notebook", "Stationery")

};

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

Product result1 = SearchUtility.linearSearch(products, "Book");

System.out.println(result1 != null ? "Found: " + result1 : "Product not found");

SearchUtility.sortByName(products);

System.out.println("=== Binary Search ===");

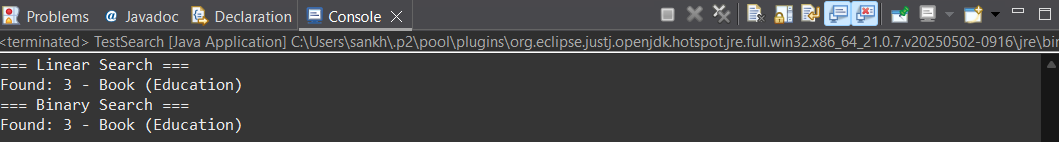
Product result2 = SearchUtility.binarySearch(products, "Book");

System.out.println(result2 != null ? "Found: " + result2 : "Product not found");

}

}

**OUTPUT :**

****

**Analysis:**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **When to Use** |
| **Linear Search** | **O(n)** | **Small or unsorted datasets** |
| **Binary Search** | **O(log n)** | **Large sorted datasets** |

**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

**SOLUTION:**

**Recursion**

Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller subproblems of the same type.  
It simplifies problems like tree traversal, factorial calculation, or Fibonacci series by handling the repeated logic internally, reducing the need for explicit loops.

**CODE:**

public class FinancialForecast {

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

if (years == 0) return principal;

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

}

public static void main(String[] args) {

double P = 10000;

double r = 0.08;

int n = 5;

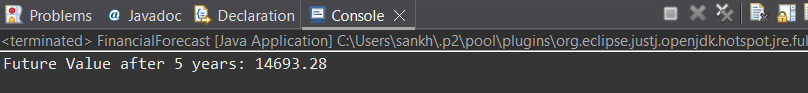
double result = futureValue(P, r, n);

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

}

}

**OUTPUT :**

****

**ANALYSIS:**

**Time Complexity**

* Without Optimization: O(n) since it performs one recursive call per year.
* Space Complexity: O(n) due to the recursive call stack.

Process to optimize the recursive solution to avoid excessive computation:

public static double futureValueIterative(double principal, double rate, int years) {

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

principal \*= (1 + rate);

}

return principal;

}