**Week1\_Algorithms\_Data Structures\_HandsOn**

**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.

**Big O notation:**

Big O Notation is a way to describe how the performance of an algorithm changes as the amount of input increases. It focuses on the growth rate of time or memory needed, rather than the exact time it takes to run. This helps us compare different algorithms based on their efficiency, especially when dealing with large inputs. Big O ignores specific hardware or programming language differences and looks only at how well an algorithm scales as data grows.

|  |  |  |
| --- | --- | --- |
| Best Case | O(1) | O(1) |
| Worst Case | O(n/2)=O(n) | O(log n) |
| Average Case | O(n) | O(log n) |

**CODE:**

import java.util.\*;

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 + ")";

}

}

public class ProductSearchProgram {

// Linear Search (case-insensitive, partial match)

public static List<Product> linearSearch(List<Product> products, String keyword) {

List<Product> result = new ArrayList<>();

for (Product product : products) {

if (product.productName.toLowerCase().contains(keyword.toLowerCase())) {

result.add(product);

}

}

return result;

}

// Binary Search (requires sorted list by name, full match only)

public static Product binarySearch(List<Product> products, String targetName) {

int low = 0;

int high = products.size() - 1;

while (low <= high) {

int mid = (low + high) / 2;

Product midProduct = products.get(mid);

int comparison = midProduct.productName.compareToIgnoreCase(targetName);

if (comparison == 0) {

return midProduct;

} else if (comparison < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

List<Product> products = new ArrayList<>();

products.add(new Product(1, "Laptop", "Electronics"));

products.add(new Product(2, "Shirt", "Clothing"));

products.add(new Product(3, "Book", "Stationery"));

products.add(new Product(4, "Phone", "Electronics"));

products.add(new Product(5, "Shoes", "Footwear"));

products.add(new Product(6, "Phone Cover", "Accessories"));

System.out.println("Choose search method:\n1. Linear Search (partial match)\n2. Binary Search (full match, sorted list)");

int method = scanner.nextInt();

scanner.nextLine();

System.out.print("Enter product name to search: ");

String keyword = scanner.nextLine();

if (method == 1) {

List<Product> found = linearSearch(products, keyword);

if (found.isEmpty()) {

System.out.println("No matching products found.");

} else {

System.out.println("Found " + found.size() + " product(s):");

for (int i = 0; i < found.size(); i++) {

System.out.println((i + 1) + ". " + found.get(i));

}

System.out.print("Select a product number (1-" + found.size() + "): ");

int choice = scanner.nextInt();

if (choice >= 1 && choice <= found.size()) {

System.out.println("You selected: " + found.get(choice - 1));

} else {

System.out.println("Invalid selection.");

}

}

} else if (method == 2) {

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

Product found = binarySearch(products, keyword);

if (found != null) {

System.out.println("Found: " + found);

} else {

System.out.println("No exact match found in binary search.");

}

} else {

System.out.println("Invalid method choice.");

}

scanner.close();

}

}

**Analysis:**

Linear search is good for small or unsorted lists but gets slower as data grows. Binary search is much faster but works only on sorted lists.  
For e-commerce platforms, binary search is better for quick and scalable product searches.

**OUTPUT:**

A screen shot of a computer program

AI-generated content may be incorrect.

**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.

**Understanding Recursive Algorithms**

Recursion is a programming technique where a function calls itself repeatedly until it reaches a base case that stops the recursion. Recursion can simplify certain problems by breaking them down into smaller sub-problems that are more manageable.

**Recursive Algorithm for Calculating Future Value**

It simplifies problems like financial forecasting, where each year’s value depends on the previous year’s value.

**CODE:**

import java.util.Scanner;

public class FinancialForecast {

// Recursive

public static double forecastRecursive(double presentValue, double growthRate, int years)

{

if (years == 0)

{

return presentValue;

}

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

}

public static void main(String[] args)

{

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the present value: ");

double presentValue = scanner.nextDouble();

System.out.print("Enter the annual growth rate : ");

double growthRate = scanner.nextDouble() / 100;

System.out.print("Enter the number of years: ");

int years = scanner.nextInt();

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

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

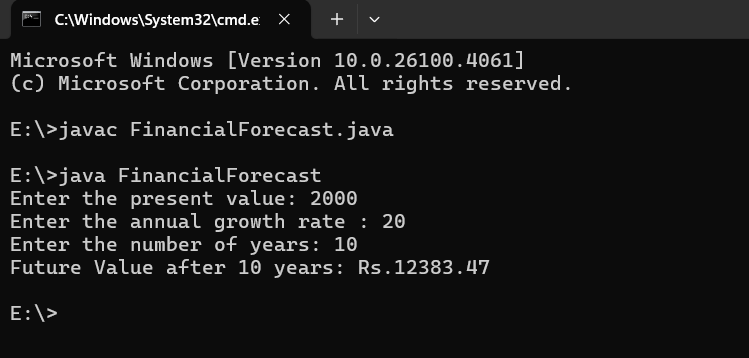
System.out.printf("Future Value after %d years: Rs.%.2f%n", years, futureValue);

scanner.close();

}

}

**OUTPUT:**

****

**Time Complexity:**

The function is called once for each year O(n) where n = years.

**Optimizing the Recursive Solution**

To optimize the recursive solution and avoid excessive computation, we can use an iterative approach .The iterative approach is generally more efficient and scalable for large inputs.