**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.

**ANSWER**

***Explain Big O notation and how it helps in analyzing algorithms :*** Big O notation is a mathematical way to describe how the performance of an algorithm changes as the size of the input grows. It gives a clear understanding of how fast or slow an algorithm is, especially as the input becomes very large. Big O focuses on the worst-case scenario and ignores constants and lower-order terms to highlight the most significant factor affecting performance.

1. Big O notation is used to describe the time complexity and space complexity of an algorithm.
2. It provides a way to express how an algorithm’s performance changes as the size of input data increases.
3. It focuses on the dominant term and ignores constants to evaluate scalability and efficiency.
4. It helps developers choose the most optimal algorithm for large datasets.
5. Big O is used to compare algorithms by their growth rates:

* Constant time → O(1)
* Logarithmic time → O(log n)
* Linear time → O(n)
* Quadratic time → O(n²), etc.

**Best Case:**

1. The scenario where the algorithm performs the fewest possible operations.
2. Example (Linear Search): Target element is at the first position.
3. Time complexity:

* Linear Search: O(1)
* Binary Search: O(1)

**Average Case:**

1. The scenario that represents the typical performance of the algorithm across many inputs.
2. It assumes the target is somewhere in the middle of the data.
3. Time complexity:

* Linear Search: O(n)
* Binary Search: O(log n)

**Worst Case:**

1. The scenario where the algorithm performs the maximum number of operations.
2. Example (Linear Search): Target is not present or at the last position.
3. Time complexity:

* Linear Search: O(n)
* Binary Search: O(log n)

**Code :-**

***File : EcommerceSearchExample/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;

    }

    public String toString()

    {

        return "[" + productId + "] " + productName + " - " + category;

    }

}

***File : EcommerceSearchExample/LinearSearch.java***

public class LinearSearch

{

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

    {

        for (Product p : products)

        {

            if (p.productName.equalsIgnoreCase(name))

            {

                return p;

            }

        }

        return null;

    }

}

***File : EcommerceSearchExample/BinarySearch.java***

import java.util.Arrays;

import java.util.Comparator;

public class BinarySearch

{

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

    {

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

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

        while (left <= right)

        {

            int mid = (left + right) / 2;

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

            if (comparison == 0)

                return products[mid];

            else if (comparison < 0)

                left = mid + 1;

            else

                right = mid - 1;

        }

        return null;

    }

}

***File: EcommerceSearchExample/SearchDemo.java***

public class SearchDemo

{

    public static void main(String[] args)

    {

        Product[] products = {

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

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

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

            new Product(104, "Watch", "Accessories")

        };

        System.out.println("Linear Search for 'Shoes':");

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

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

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

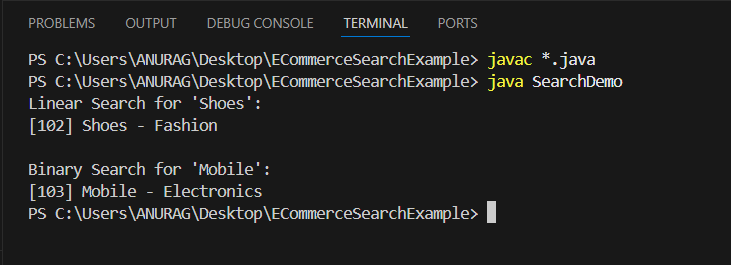
        Product found2 = BinarySearch.binarySearch(products, "Mobile");

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

    }

}

**Output:-**

****

For large, mostly static product lists : Binary Search is ideal due to faster performance (O(log n))  
For dynamic, unsorted data : Linear Search may be useful if no other options are available.

**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.

**ANSWER**

**Explain the concept of recursion and how it can simplify certain problems:** Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem**.**

Recursion is useful in problems that have a hierarchical or repetitive structure. It can make code cleaner and more intuitive, especially for:

* Tree traversals
* Divide and conquer algorithms
* Mathematical computations
* Backtracking problems, etc.

**Code:-**

**File: FinancialForecasting/FinancialForecast.java**

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 principal = 10000;

        double rate = 0.05;

        int years = 5;

        double result = futureValue(principal, rate, years);

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

    }

}

**File: FinancialForecasting/Main.java**

public class Main

{

    public static void main(String[] args)

    {

        double principal = 15000;

        double rate = 0.06;

        int years = 7;

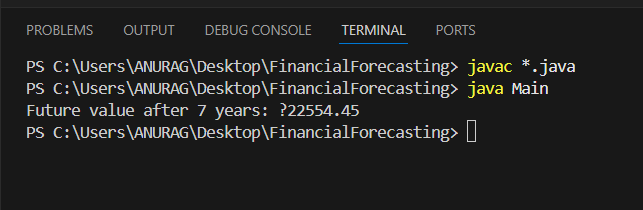
        double result = FinancialForecast.futureValue(principal, rate, years);

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

    }

}

**Output:-**

****

**Time Complexity:**

* The recursion runs n times. Therefore it has O(n) time complexity.
* Each call does constant work and has n number of recursive calls.

**Optimization:**

Recursion can be inefficient if it repeats calculations. We can optimize this as follows:

* Use memoization to store intermediate results using dynamic programming approach.
* Convert to iterative solution for better space performance (recursion using call stack).