**Ex2)E-commerce Platform Search Function:**

Asymptotic notations:

Big O Notation:

Big O describes the worst-case time complexity of an algorithm.

It gives an upper bound on how an algorithm performs as the input size grows.

Examples:

O(1) = Constant time

O(n) = Linear time

O(log n) = Logarithmic time

O(n log n) = Linearithmic time

O(n²) = Quadratic time

Best, average, and worst-case scenarios for search operations:

Linear Search – Best: O(1), Average: O(n), Worst: O(n)

Binary Search – Best: O(1), Average: O(log n), Worst: O(log n)

**Setup**:

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 + "]";

    }

}

ProductSearch.java:

import java.util.Arrays;

import java.util.Comparator;

public class ProductSearch {

// Linear Search

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

        for (Product p : products) {

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

                return p;

            }

        }

        return null;

}

// Binary Search

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

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

        while (low <= high) {

            int mid = (low + high) / 2;

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

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

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

            else high = mid - 1;

        }

return null;

    }

    // Test method

    public static void main(String[] args) {

        Product[] products = {

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

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

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

            new Product(104, "Book", "Education"),

            new Product(105, "Watch", "Accessories")

};

// Linear Search Test

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

        Product result1 = linearSearch(products, "Phone");

        System.out.println(result1 != null ? result1 : "Not found");

  // Sort for Binary Search

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

        // Binary Search Test

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

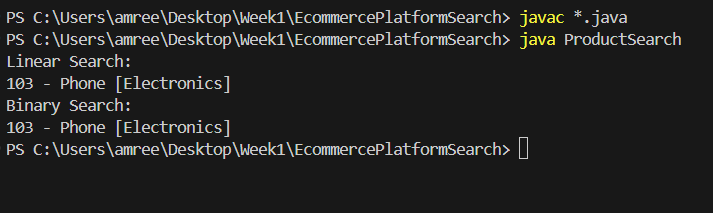
        Product result2 = binarySearch(products, "Phone");

        System.out.println(result2 != null ? result2 : "Not found");

    }

}

**Output:**



**Analysis:**

Linear search has a time complexity of O(1) in the best case, O(n) in the average case, and O(n) in the worst case. Binary search, on the other hand, has a time complexity of O(1) in the best case and O(log n) in both the average and worst cases. However, binary search requires the array to be sorted beforehand.

**Binary Search is Better Option for E-commerce platform:**

For an e-commerce platform, binary search is generally more suitable because it provides much faster search performance, especially when dealing with large datasets. If the product list does not change frequently and can be kept sorted, binary search is ideal for ensuring a fast and responsive user experience. Linear search may still be acceptable for small datasets or when sorting is not feasible due to frequent updates, but it is not efficient for large-scale search operations.

**Ex7)Financial Forecasting**

**Recursion:**

Recursion is a programming technique where a function calls itself to solve a smaller version of the original problem.

Key concepts:

* Base Case: The condition under which recursion stops.
* Recursive Case: The part of the function that breaks the problem down into smaller instances and makes the recursive call.

It simplifies problems that have repetitive or hierarchical structures, such as:

Tree traversal

Factorials or Fibonacci sequence

**Recursive Algorithm:**

A recursive algorithm is an algorithm that solves a problem by calling itself with a smaller subproblem until it reaches a base case (a stopping condition).

**Recursive Method for Future Value:**

To calculate future value recursively:

FV(n) = (1 + r) × FV(n - 1)

Base case: FV(0) = P

**Implementation:**

**FinancialForecast.java:**

public class FinancialForecast {

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

if (periods == 0) {

            return principal;

        }

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

    }

public static void main(String[] args) {

        double principal = 1000.0;

        double rate = 0.05;

        int periods = 5;

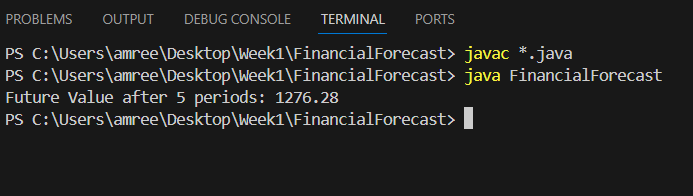
double result = futureValue(principal, rate, periods);

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

    }

}

**Output:**



**Analysis:**

Time complexity: O(n)

Because for each period, the method makes one recursive call. This means the function is called n times before reaching the base case.

Space complexity: O(n)

It is due to the call stack created by recursive calls. Each function call remains in memory until the base case is reached and the results begin to return back up the call chain.

**Optimization:**

To optimize this recursive solution, we can convert it to an iterative approach. This avoids the overhead of the recursive call stack and improves performance, especially for large input values. An iterative solution uses a loop to multiply the result by (1 + rate) for each period, which achieves the same outcome in constant space.

**Optimized code:**

public class FinancialForecast {

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

double result = principal;

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

result \*= (1 + rate);

}

return result;

}

public static void main(String[] args) {

double principal = 1000.0;

double rate = 0.05;

int periods = 5;

double result = futureValueIterative(principal, rate, periods);

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

}

}