**Recursion Problems: Step-by-Step Analysis and Big-O Complexity**

### Introduction

This document explains the Java program that performs multiple recursive and iterative computations. The program includes functions for:

* Finding the minimum value in an array
* Calculating the average of an array
* Checking if a number is prime
* Computing factorial recursively
* Finding Fibonacci numbers using recursion
* Computing power using recursion
* Checking if a string consists only of digits
* Calculating the binomial coefficient
* Finding the greatest common divisor (GCD) using the Euclidean algorithm
* Printing an array in reverse order recursively

**Task 1: Finding Minimum in an Array**

### Code Explanation

#### Importing Scanner Class

import java.util.Scanner;

The Scanner class is imported to take user input.

#### Main Class Definition

public class RecursionProblems {

Defines the main class RecursionProblems.

#### Main Method

public static void main(String[] args) {

This is the entry point of the program where all user input is taken and processed.

#### Taking User Input

The program first takes input for the number of elements in an array and then fills the array.

System.out.print("Enter number of elements in array: ");

int n = scanner.nextInt();

int[] arr = new int[n];

System.out.println("Enter array elements: ");

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

arr[i] = scanner.nextInt();

}

Next, it takes input for different computations such as prime checking, factorial calculation, Fibonacci sequence, power calculation, binomial coefficient, and GCD.

#### Execution Time Measurement

Each function call is wrapped with execution time measurement:

long startTime, endTime;

startTime = System.nanoTime();

System.out.println("Minimum: " + findMin(arr, n));

endTime = System.nanoTime();

System.out.println("Time taken: " + (endTime - startTime) + " ns");

**Task 1: Finding Min**

**Method: findMin(int[] arr, int n)**

**Steps:**

1. Initialize min as the first element of the array.
2. Iterate through the array comparing each element with min.
3. If an element is smaller than min, update min.
4. Return min.

**Big-O Complexity Calculation:**

* We iterate through the array once.
* Each iteration takes O(1) time for comparison.
* For n elements, the total complexity is: T(n)=O(n)T(n) = O(n)

**Task 2: Finding Average of an Array**

**Method: findAverage(int[] arr, int n)**

**Steps:**

1. Initialize sum to 0.
2. Loop through the array, adding each element to sum.
3. Compute the average by dividing sum by n.
4. Return the average.

**Big-O Complexity Calculation:**

* Summation of n elements takes O(n) time.
* Division operation takes O(1) time.
* Therefore: T(n)=O(n)T(n) = O(n)

**Task 3: Checking if a Number is Prime**

**Method: isPrime(int n, int i)**

**Steps:**

1. If n <= 2, return whether n is 2.
2. If n is divisible by i, return false.
3. If i^2 > n, return true.
4. Recursively check divisibility with i + 1.

**Big-O Complexity Calculation:**

* The function checks divisibility up to n\sqrt{n}.
* Since we iterate approximately n\sqrt{n} times: T(n)=O(n)T(n) = O(\sqrt{n})

**Task 4: Factorial Calculation**

**Method: factorial(int n)**

**Steps:**

1. If n is 0 or 1, return 1.
2. Otherwise, return n \* factorial(n - 1).

**Big-O Complexity Calculation:**

* Recursion depth is n.
* Each call takes O(1) time.
* Therefore: T(n)=O(n)T(n) = O(n)

**Task 5: Fibonacci Calculation**

**Method: fibonacci(int n)**

**Steps:**

1. If n is 0, return 0.
2. If n is 1, return 1.
3. Otherwise, return fibonacci(n - 1) + fibonacci(n - 2).

**Big-O Complexity Calculation:**

* Each call results in two new calls.
* This forms a binary recursion tree with depth n.
* Total operations grow exponentially: T(n)=O(2n)T(n) = O(2^n)

**Task 6: Power Calculation**

**Method: power(int a, int n)**

**Steps:**

1. If n is 0, return 1.
2. Otherwise, return a \* power(a, n - 1).

**Big-O Complexity Calculation:**

* Recursion depth is n.
* Each call takes O(1) time.
* Therefore: T(n)=O(n)T(n) = O(n)

**Task 7: Printing an Array in Reverse**

**Method: printReverse(int[] arr, int index)**

**Steps:**

1. If index < 0, stop.
2. Print arr[index].
3. Recursively call with index - 1.

**Big-O Complexity Calculation:**

* Recursion depth is n.
* Each call takes O(1) time.
* Therefore: T(n)=O(n)T(n) = O(n)

**Task 8: Checking If a String Contains Only Digits**

**Method: isAllDigits(String s, int index)**

**Steps:**

1. If index == s.length(), return true.
2. If s[index] is not a digit, return false.
3. Recursively check the next index.

**Big-O Complexity Calculation:**

* Recursion depth is n.
* Each call takes O(1) time.
* Therefore: T(n)=O(n)T(n) = O(n)

**Task 9: Binomial Coefficient Calculation**

**Method: binomialCoefficient(int n, int k)**

**Steps:**

1. If k == 0 or k == n, return 1.
2. Otherwise, return binomialCoefficient(n-1, k-1) + binomialCoefficient(n-1, k).

**Big-O Complexity Calculation:**

* Each call generates two more calls.
* This forms an exponential growth tree.
* Therefore: T(n)=O(2n)T(n) = O(2^n)

**Task 10: GCD Calculation (Euclidean Algorithm)**

**Method: gcd(int a, int b)**

**Steps:**

1. If b == 0, return a.
2. Recursively compute gcd(b, a % b).

**Big-O Complexity Calculation:**

* The problem size reduces significantly in each step.
* The number of recursive calls is O(log⁡min⁡(a,b))O(\log \min(a, b)). T(n)=O(log⁡n)T(n) = O(\log n)