C-DAC Mumbai

Date 25/09/2024

Subject: Algorithm and Data Structure

Assignment 1

Solve the assignment with the following thing to be added in each question.
-Program
-Flow chart
-Explanation
-Output
-Time and Space complexity

1. Armstrong Number

Problem: Write a Java program to check if a given number is an Armstrong number.

Test Cases:
Input:
153
Output: true
Input: 123
Output: false

Solution:

-Program

```
import java.util.Scanner;

public class ArmstrongNumber {
    public static boolean isArmstrong(int number){
        int originalNumber = number, result = 0, digits = 0;

        while (originalNumber != 0){
            originalNumber /= 10;
            digits++;
        }
        originalNumber = number;

        while(originalNumber != 0){
            int remainder = originalNumber % 10;
            result += Math.pow(remainder, digits);
            originalNumber /= 10;
        }
        return result == number;
    }
```

```
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter a number: ");
        int number = scanner.nextInt();

        if(isArmstrong(number)){
            System.out.println("True");
        }else{
            System.out.println("False");
        }
        scanner.close();
    }
}
```

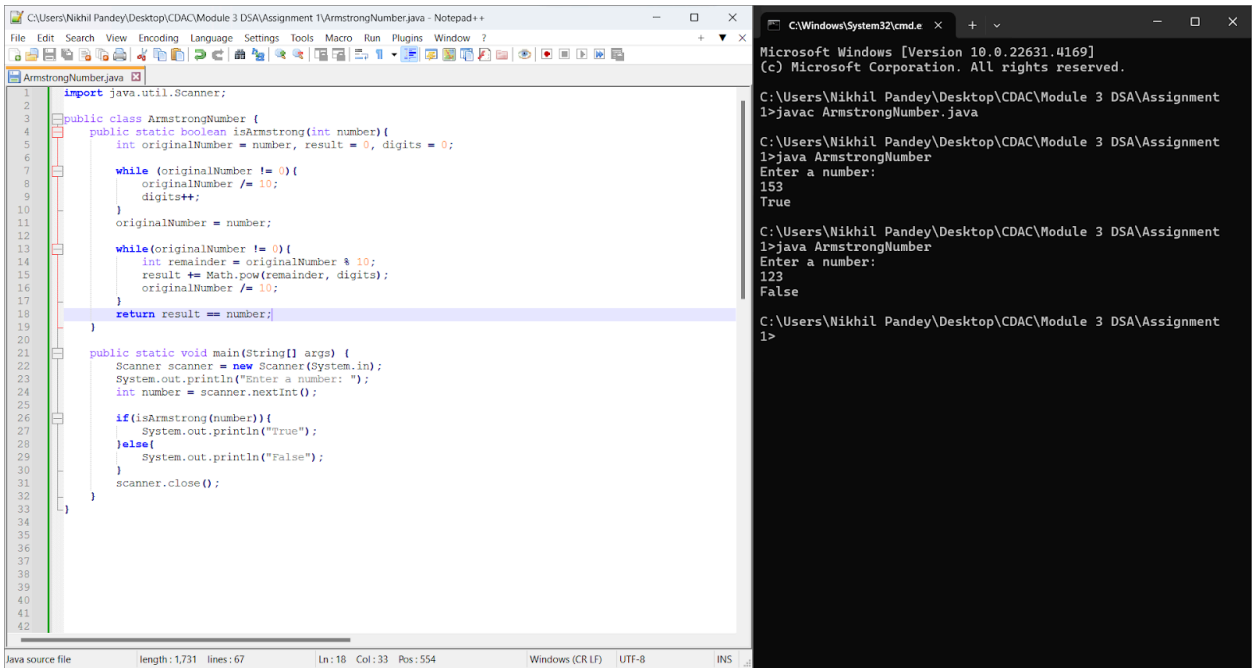
-Flow Chart

```
/*
            Start
              |
        Enter number (n)
              |
        Count digits (d)
              |
    Calculate sum of each digit ^ d
              |
      Is sum == number (n)?
                /     \
               Yes     No
               /         \
         "Armstrong"    "Not Armstrong"
               \         /
                    |
                  End
*/
```

-Explanation

The program first counts the number of digits in the input number.
Then it calculates the sum of each digit raised to the power of the total number of digits.
Finally, it checks if this calculated sum is equal to the original number. If yes, the number is an Armstrong number; otherwise, it is not.
Armstrong Number Definition: A number is called an Armstrong number if the sum of its digits raised to the power of the number of digits equals the number itself. For example, 153 is an Armstrong number because
1^3+5^3+3^3=153.

-Output



-Time and Space complexity

Time Complexity:
The time complexity of this program is **O(d)** , where d is the number of digits in the input number. This is because we iterate over each digit twice: once for counting the digits and once for calculating the sum of powers.

Space Complexity:
The space complexity of this program is **O(1)** since the space used by the program does not depend on the size of the input, aside from a few integer variables.

2. Prime Number

Problem: Write a Java program to check if a given number is prime.

Test Cases:
Input: 29
Output: true
Input: 15
Output: false

Solution:

-Program

```
import java.util.Scanner;

public class PrimeNumber {
    public static boolean isPrime(int number) {
        if(number <= 1){
            return false;
        }
        for(int i = 2; i < Math.sqrt(number); i++) {
            if(number % i == 0){
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter a number: ");
        int number = scanner.nextInt();

        if(isPrime(number)){
            System.out.println("True");
        }else{
            System.out.println("False");
        }

        scanner.close();
    }
}
```

-Flow Chart

```
/*
          Start
            |
       Enter number (n)
            |
     Is n <= 1? ---> Yes --> "Not Prime"
            |
            No
            |
   Check divisibility from 2 to √n
      /            \
  Divisible?       No divisor
    /                \
 "Not Prime"        "Prime Number"
            |
          End

*/
```

-Explanation

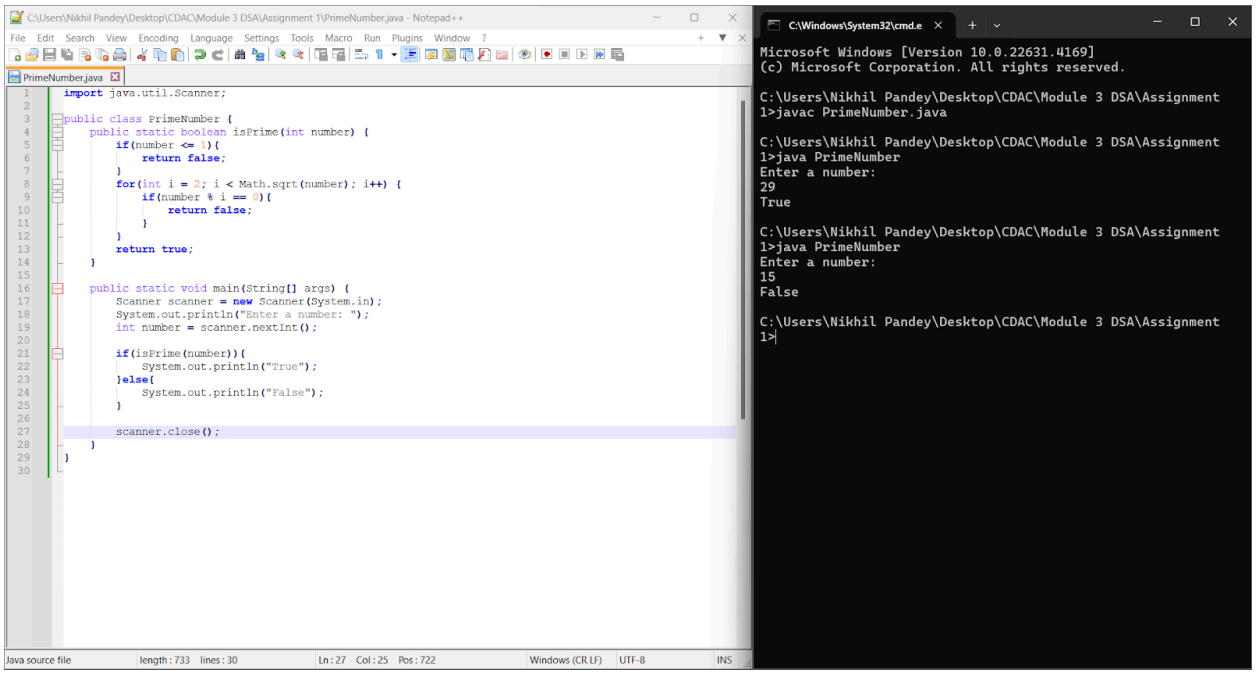
- A prime number is a natural number greater than 1 that is not divisible by any number other than 1 and itself.

- The program checks if the input number is less than or equal to 1, in which case it is not a prime number.

- If the number is greater than 1, the program checks divisibility by numbers from 2 up to the square root of the number. If it finds a divisor, the number is not prime.

- If no divisors are found, the number is prime.

-Output



-Time and Space complexity

Time Complexity:

- The time complexity is **O(√n)** because we only check divisibility up to the square root of the number.

Space Complexity:

- The space complexity is **O(1)** as the program only uses a few integer variables.

3. Factorial

Problem: Write a Java program to compute the factorial of a given number.

Test Cases:
Input: 5
Output: 120
Input: 0
Output: 1

Solution:

-Program

```
import java.util.Scanner;

public class Factorial {
   public static long factorial(int number) {
      long result = 1;
   for(int i = 2; i <= number; i++) {
      result *= i;
   }
   return result;
   }


public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.println("Enter a number");
      int number = scanner.nextInt();

      long fact = factorial(number);
      System.out.println(fact);

      scanner.close();
   }
}
```

-Flow Chart

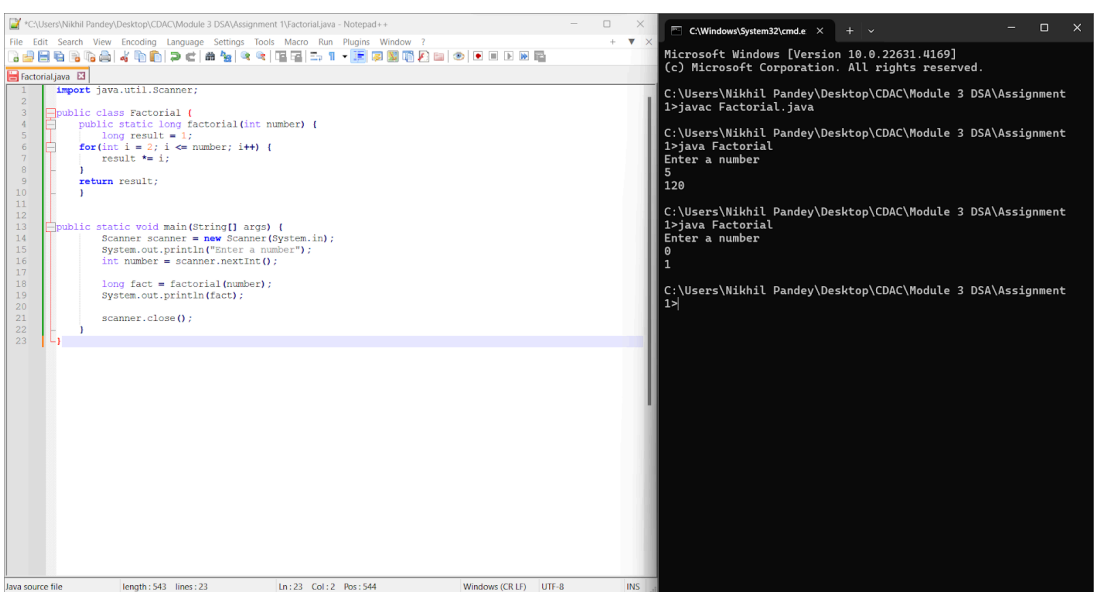
```
/*
        Start
          |
     Enter number (n)
          |
    Initialize result = 1
          |
    For i = 2 to n
          |
    result = result * i
          |
     Output result
          |
        End

*/
```

-Explanation

Factorial of a number n is the product of all positive integers less than or equal to n. Mathematically, it is denoted as $n!$ and is computed as
n!=n×(n−1)×(n−2)×...×1.
The program initializes result to 1, then multiplies it by each integer from 2 to the input number.
If the input number is 0 or 1, the factorial is 1 (by definition).

-Output



-Time and Space complexity

Time Complexity:

- The time complexity is **O(n)**, where n is the input number, since we loop from 2 to the number to calculate its factorial.

Space Complexity:

- The space complexity is **O(1)** as the program only uses a few variables to store the result and the loop counter.

4. Fibonacci Series

Problem: Write a Java program to print the first n numbers in the Fibonacci series.

Test Cases:
Input: n = 5
Output: [0, 1, 1, 2, 3]
Input: n = 8
Output: [0, 1, 1, 2, 3, 5, 8, 13]

Solution:
-Program

```
import java.util.Scanner;
import java.util.ArrayList;

public class FibonacciSeries{
   public static ArrayList <Integer> getFibonacciSeries(int n){
                ArrayList<Integer> series = new ArrayList<>();

                if(n <= 0)return series;
                series.add(0);
                if (n == 1) return series;

                int first = 0, second = 1;
                series.add(second);

                for(int i = 2; i < n; i++){
                        int next = first + second;
                        series.add(next);
                        first = second;
                        second = next;
                }
                return series;
   }
```

```
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
                System.out.print("Enter the value of n: ");
                int n = scanner.nextInt();

                ArrayList<Integer> fibonacciSeries = getFibonacciSeries(n);
                System.out.println("Fibonacci Series: " + fibonacciSeries);

                scanner.close();
    }
}
```

-Flow Chart

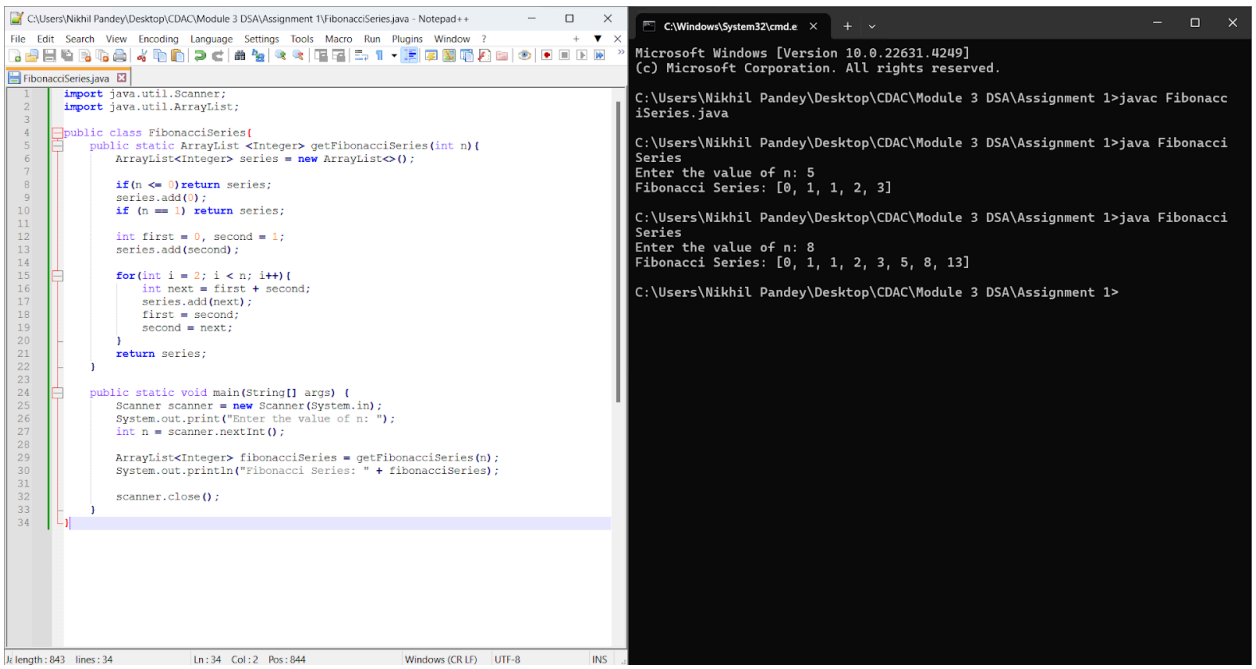
```
/*
          Start
            |
       Enter value of n
            |
   Is n <= 0? ---> Yes --> Return empty series
            |
            No
            |
    Initialize first = 0, second = 1
            |
  Add first and second to series
            |
   Loop i from 3 to n
            |
  Calculate next = first + second
            |
  Add next to series
            |
  Update first = second, second = next
            |
         End
*/
```

-Explanation

Fibonacci Series: A sequence where each number is the sum of the two preceding ones, starting from 0 and 1.

The program initializes the first two Fibonacci numbers (`first = 0` and `second = 1`), adds them to the list, and computes successive numbers by adding the previous two.
If **n** is 0 or less, the program returns an empty list, and if **n = 1**, it returns only the first Fibonacci number.
The program iterates until the **n**th Fibonacci number is generated and printed.

-Output



-Time and Space complexity

Time Complexity: The time complexity is **O(n)**, where n is the input number. The program iterates n times to calculate each Fibonacci number.

Space Complexity: The space complexity is **O(n)** because we store the Fibonacci series in a list of size n.

5. Find GCD
Problem: Write a Java program to find the Greatest Common Divisor (GCD) of two numbers.

Test Cases:
Input: a = 54, b = 24
Output: 6
Input: a = 17, b = 13
Output: 1

Solution:
-Program

```
import java.util.Scanner;

public class GCD {
    public static int findGCD(int a, int b) {
        if(b == 0) {
            return a;
        }
        return findGCD(b, a % b);
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter first number (a): ");
        int a = scanner.nextInt();
        System.out.print("Enter second number (b): ");
        int b = scanner.nextInt();

        int gcd = findGCD(a, b);
        System.out.println("GCD of " + a + " and " + b + " is: " + gcd);

        scanner.close();

    }
}
```

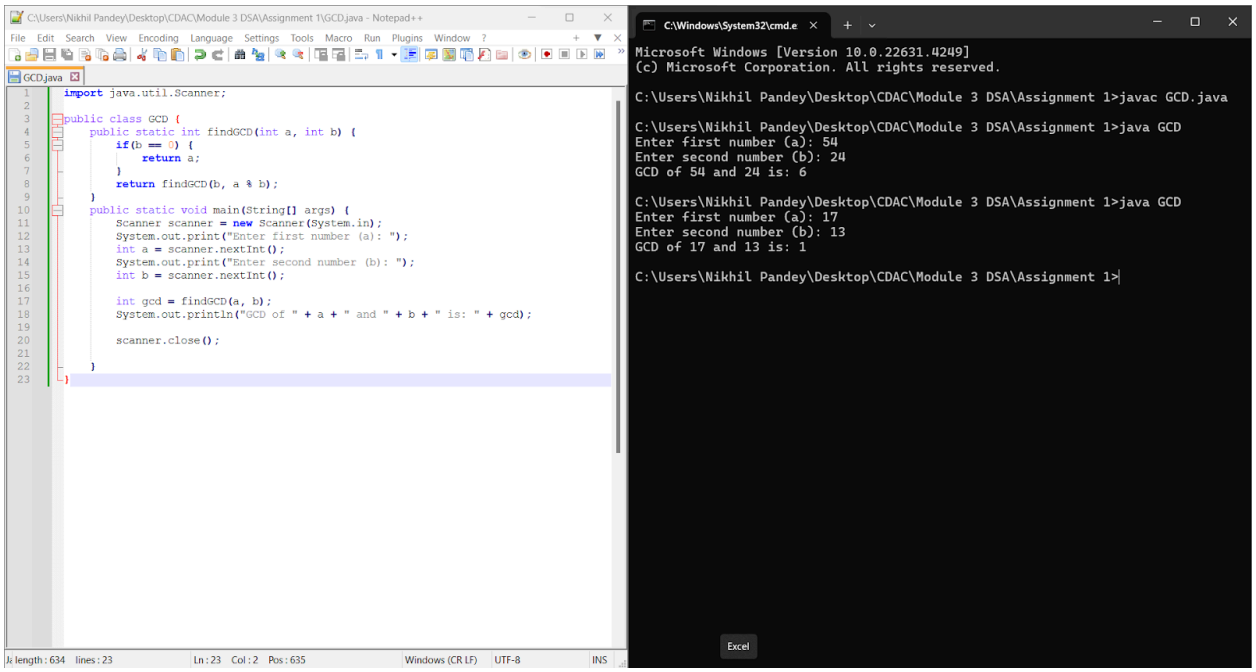
-Flow Chart

```
/*
        Start
          |
   Enter numbers a, b
          |
  Is b = 0? ---> Yes --> GCD = a
          |
         No
          |
   Calculate a % b
          |
     a = b, b = a % b
          |
   Repeat until b = 0
          |
      Output GCD
          |
         End
*/
```

-Explanation

Euclidean Algorithm: The GCD of two numbers a and b is calculated by repeatedly applying the equation GCD(a, b) = GCD(b, a % b) until b becomes zero. At that point, a is the GCD.
The program recursively calls the findGCD() function until the second number b becomes zero, and returns the first number a as the GCD.
The input numbers are taken from the user, and the result is displayed.

-Output



-Time and Space complexity

Time Complexity:
The time complexity is **O(log(min(a, b)))** because with each step, the value of b is reduced by the modulus operation until b becomes zero.

Space Complexity:
The space complexity is **O(log(min(a, b)))** due to the recursive calls in the Euclidean algorithm. If an iterative version is used, the space complexity would be **O(1)**.

6. Find Square Root
Problem: Write a Java program to find the square root of a given number (using integer approximation).

Test Cases:
Input: x = 16
Output: 4
Input: x = 27
Output: 5

Solution:
<u>-Program</u>

```
import java.util.Scanner;

public class SquareRoot {
    public static int findSquareRoot(int x){
        if(x < 0){
            System.out.println("Square root of negative numbers is not supported.");
            return -1;
        }
        if (x == 0 || x == 1) {
            return x;
        }
        int start = 1, end = x, ans = 0;

        while (start <= end){
            int mid = (start + end) / 2;
            if (mid * mid == x) {
                return  mid;
            }
            if (mid * mid < x) {
                start = mid + 1;
                ans = mid;
            } else {
                end = mid - 1;
            }
        }
        return ans;
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter a number: ");
        int x = scanner.nextInt();
```

```
        int sqrt = findSquareRoot(x);
        if (sqrt != 1) {
            System.out.println("The integer square root of " + x + " is: " + sqrt);
        }
        scanner.close();
    }
}
```

-Flow Chart

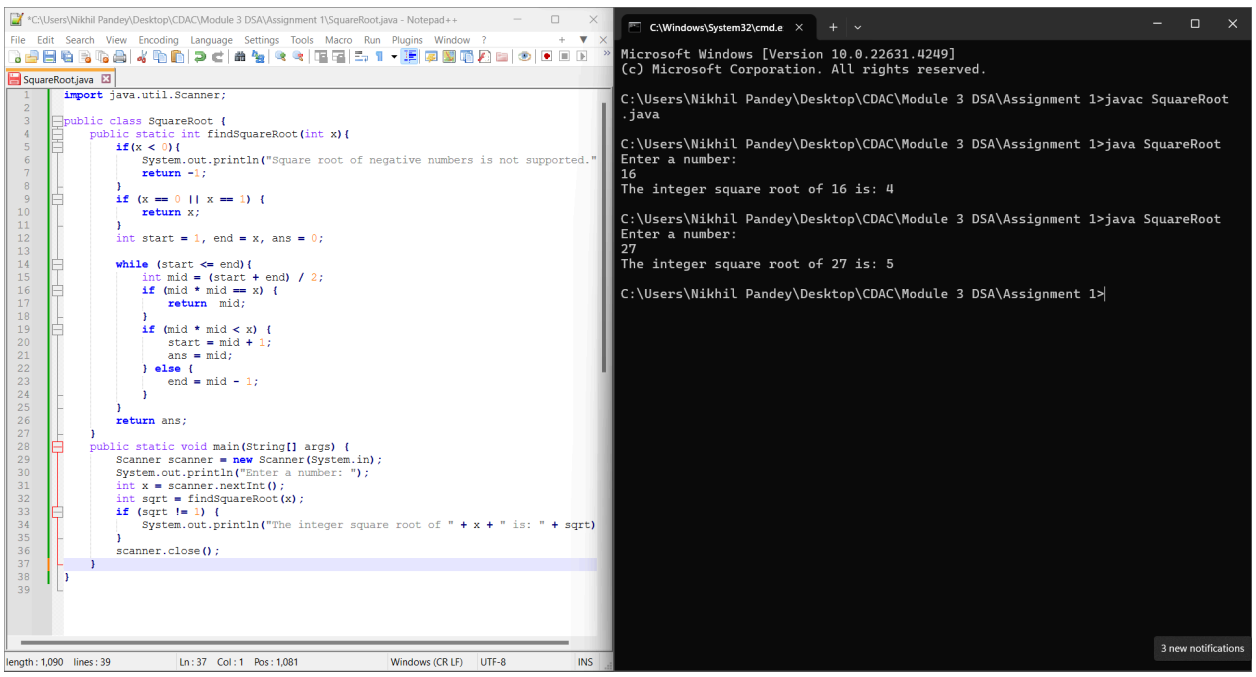
```
/*
            Start
              |
        Enter number x
              |
        Is x < 0? ---> Yes --> "Invalid Input"
              |
              No
              |
   Is x = 0 or x = 1? --> Yes --> return x
              |
              No
              |
   Initialize start = 1, end = x
              |
       While start <= end
              |
       Calculate mid = (start + end) / 2
              |
    Is mid^2 = x? --> Yes --> return mid
              |
              No
              |
   Is mid^2 < x? --> Yes --> start = mid + 1, ans = mid
              |
              No --> end = mid - 1
              |
        Return ans (closest approximation)
              |
            End

*/
```

-Explanation

The program uses binary search to find the square root. The search space is between 1 and the given number.
At each step, it checks if the middle number (`mid`) squared is equal to the given number. If yes, it returns the exact square root.
If `mid * mid` is less than the number, the search continues in the right half of the search space; otherwise, it continues in the left half.
The variable `ans` keeps track of the closest integer approximation of the square root.
If the input is less than 0, the program returns an error message since the square root of negative numbers is not supported.

-Output



-Time and Space complexity

Time Complexity:
The time complexity is **O(log(x))**, where x is the input number. This is because we are using binary search to find the square root.

Space Complexity:
The space complexity is **O(1)** as the program only uses a few variables (start, end, mid, ans) regardless of the size of the input.

7. Find Repeated Characters in a String
Problem: Write a Java program to find all repeated characters in a string.

Test Cases:
Input: "programming"
Output: ['r', 'g', 'm']
Input: "hello"
Output: ['l']

Solution:
-Program

```
import java.util.HashMap;
import java.util.ArrayList;
import java.util.Scanner;

public class RepeatedCharacters {
    public static ArrayList<Character> findRepeatedChars(String str){
        HashMap<Character, Integer> charCountMap = new HashMap<>();
        ArrayList<Character> repeatedChars = new ArrayList<>();

        for(int i = 0; i < str.length(); i++){
            char currentChar = str.charAt(i);
            charCountMap.put(currentChar, charCountMap.getOrDefault(currentChar,0) + 1);
        }
        for(char key : charCountMap.keySet()){
            if (charCountMap.get(key) > 1) {
                repeatedChars.add(key);
            }
        }
        return repeatedChars;
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter a string: ");
        String input = scanner.nextLine();

        ArrayList<Character> result = findRepeatedChars(input);
        System.out.println("Repeated Characters: " + result);

        scanner.close();
    }
}
```

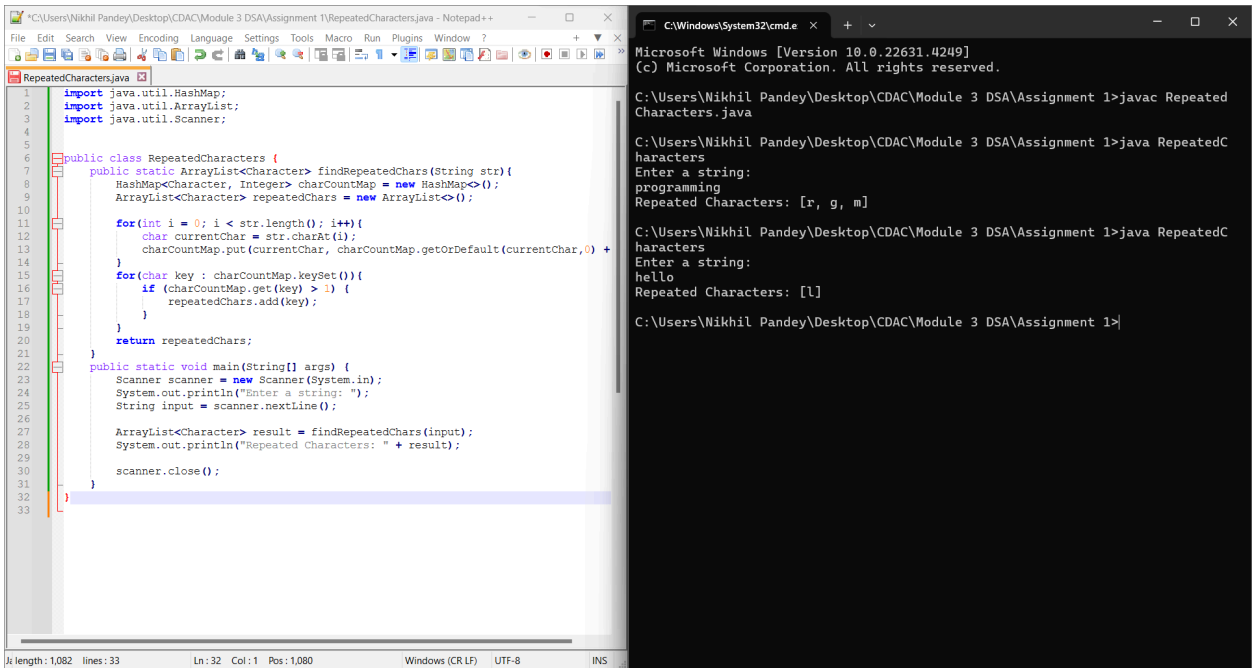
-Flow Chart

```
/*
          Start
            |
        Enter string str
            |
      Initialize charCountMap
            |
      Loop through string str
            |
   Add characters to charCountMap with counts
            |
      Loop through charCountMap
            |
  If count > 1, add to repeatedChars list
            |
    Output repeated characters
            |
          End
*/
```

-Explanation

The program uses a HashMap to keep track of the frequency of each character in the input string.
It iterates through the string, updating the count of each character in the map.
After populating the map, the program checks which characters have a count greater than 1 (i.e., repeated characters) and adds them to a list of repeated characters.
The program then prints the repeated characters as output.

-Output



-Time and Space complexity

Time Complexity:
The time complexity is **O(n)**, where n is the length of the input string. The program iterates through the string once to count the characters and once more to find the repeated characters.

Space Complexity:
The space complexity is **O(n)** due to the HashMap that stores the frequency of each character and the list that stores the repeated characters.

8. First Non-Repeated Character
Problem: Write a Java program to find the first non-repeated character in a string.

Test Cases:
Input: "stress"
Output: 't'
Input: "aabbcc"
Output: null

Solution:
<u>-Program</u>

```
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Scanner;

public class FirstNonRepeatedCharacter {
    public static Character findFirstNonRepeatedChar(String str){
        LinkedHashMap<Character, Integer> charCountMap = new LinkedHashMap<>();

        for(int i = 0; i < str.length(); i++){
            char currentChar = str.charAt(i);
            charCountMap.put(currentChar, charCountMap.getOrDefault(currentChar, 0) + 1);
        }
        for(Map.Entry<Character, Integer> entry : charCountMap.entrySet()){
            if (entry.getValue() == 1){
                return entry.getKey();
            }
        }
        return null;
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter a string: ");
        String input = scanner.nextLine();
        Character result = findFirstNonRepeatedChar(input);
            if (result != null) {
                System.out.println("The first non-repeated character is : " + result);
            } else {
                System.out.println("No non-repeated character found.");
            }
            scanner.close();
    }
}
```

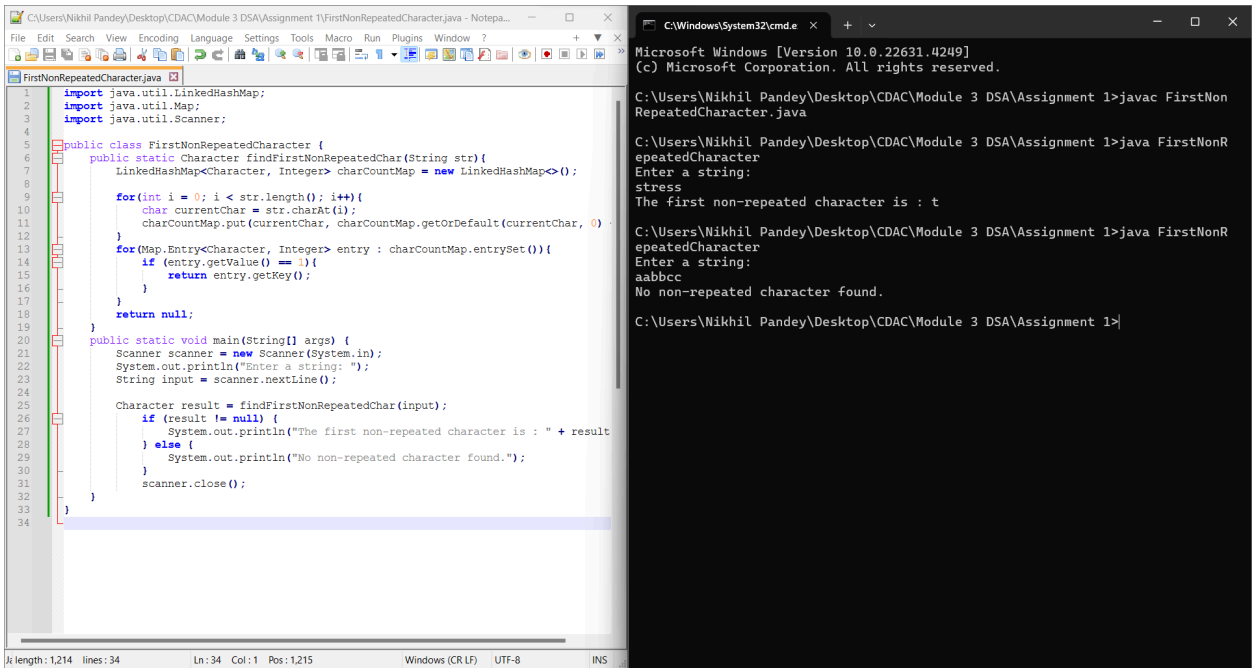
-Flow Chart

```
/*
          Start
            |
        Enter string str
            |
   Initialize LinkedHashMap to store character counts
            |
   Loop through string and update charCountMap
            |
   Loop through charCountMap to find first character with count = 1
            |
  If found, output first non-repeated character
            |
   If none found, output "No non-repeated character found"
            |
          End
*/
```

-Explanation

The program uses a LinkedHashMap to store the count of each character while maintaining the order of insertion.
The string is first looped through to update the count of each character in the map.
The program then loops through the map entries to find the first character that has a count of 1 (i.e., a non-repeated character).
If no such character is found, the program returns null and prints a message indicating that no non-repeated character exists.

-Output



-Time and Space complexity

Time Complexity:
The time complexity is **O(n)**, where n is the length of the input string. The program loops through the string twice: once to count the characters and once to find the first non-repeated character.

Space Complexity:
The space complexity is **O(n)** due to the LinkedHashMap used to store the character counts.

9. Integer Palindrome
Problem: Write a Java program to check if a given integer is a palindrome.

Test Cases:
Input: 121
Output: true
Input: -121
Output: false

Solution:
-Program

```
import java.util.Scanner;

public class IntegerPalindrome {
    public static boolean isPalindrome(int num){
        if (num < 0) {
            return false;
        }
        int originalNum = num;
        int reverseNum = 0;

        while (num != 0) {
            int lastDigit = num % 10;
            reverseNum = reverseNum * 10 + lastDigit;
            num = num / 10;
        }
        return originalNum == reverseNum;
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter an integer: ");
        int num = scanner.nextInt();

        boolean result = isPalindrome(num);
        if (result) {
            System.out.println("True");
        } else {
            System.out.println("False");
        }
        scanner.close();
    }
}
```

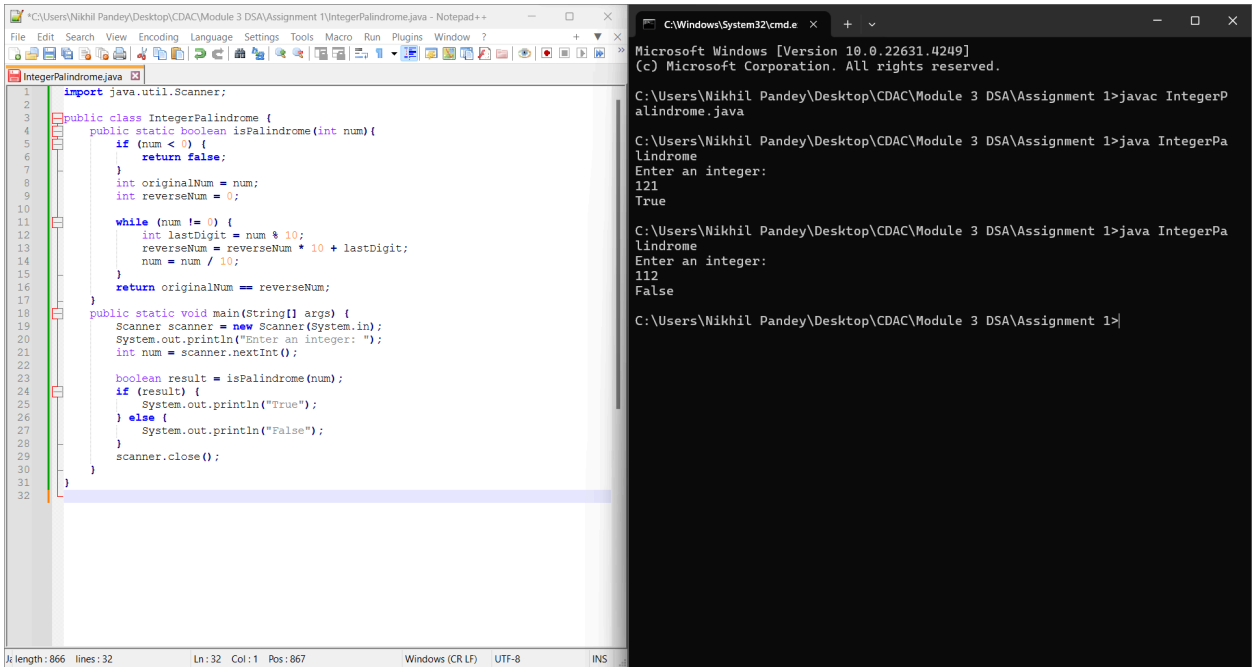
-Flow Chart

```
/*
          Start
            |
        Enter integer num
            |
       Is num < 0? ---> Yes --> Output: Not a palindrome
            |
            No
            |
      Initialize reversedNum = 0, originalNum = num
            |
      While num != 0
            |
     Get lastDigit = num % 10
            |
     Update reversedNum = reversedNum * 10 + lastDigit
            |
     Update num = num / 10
            |
   Check if originalNum == reversedNum
            |
  If true --> Output: Palindrome
  If false --> Output: Not a palindrome
            |
          End
*/
```

-Explanation

Negative numbers are immediately ruled out as non-palindromes because a palindrome must read the same forward and backward, and negative signs make this impossible.
The program reverses the digits of the given integer. It extracts each digit using the modulo operator (%), constructs the reversed number, and compares it with the original integer.
If the original number equals its reversed version, it is considered a palindrome; otherwise, it is not.

-Output



-Time and Space complexity

Time Complexity:
The time complexity is **O(d)**, where d is the number of digits in the integer. The program iterates over each digit to reverse the number.

Space Complexity:
The space complexity is **O(1)**, since the program only uses a few integer variables (num, reversedNum, lastDigit, and originalNum) regardless of the size of the input number.

10. Leap Year
Problem: Write a Java program to check if a given year is a leap year.

Test Cases:
Input: 2020
Output: true
Input: 1900
Output: false

Solution:
-Program

```
import java.util.Scanner;

public class LeapYear {

    public static boolean isLeapYear(int year) {

        if (year % 4 == 0) {
            if (year % 100 == 0) {
                return year % 400 == 0;
            }
            return true;
        }
        return false;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a year: ");
        int year = scanner.nextInt();

        boolean result = isLeapYear(year);
        if (result) {
            System.out.println("True");
        } else {
            System.out.println("False");
        }

        scanner.close();
    }
}
```

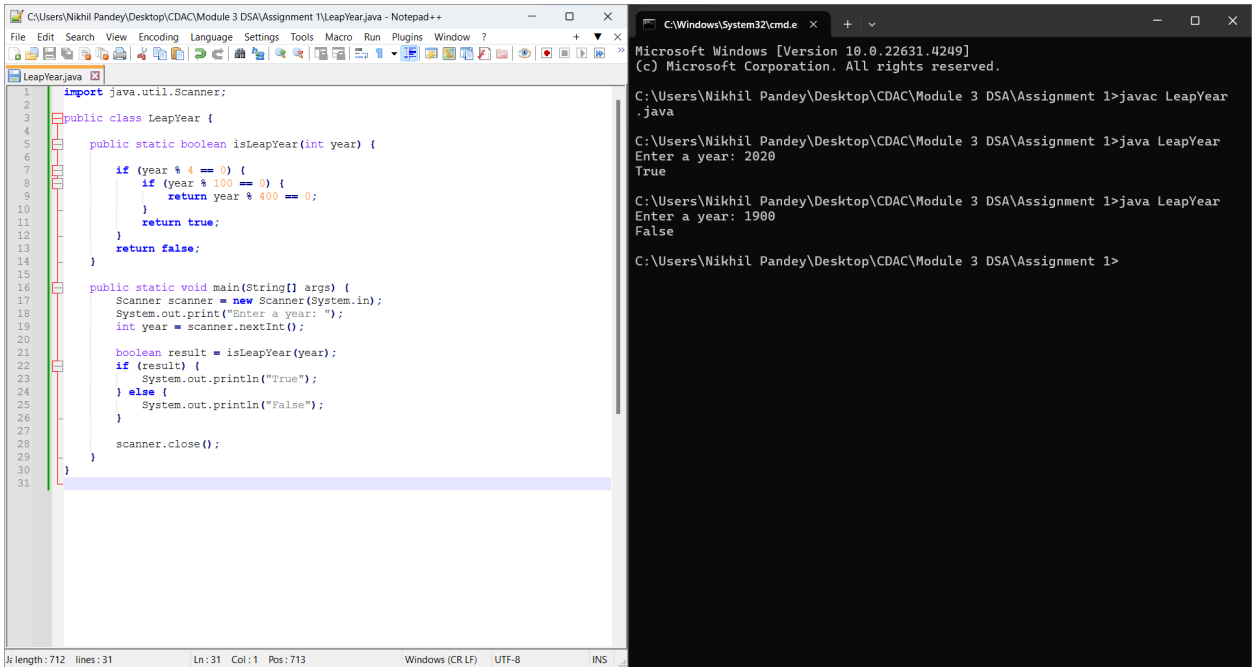
-Flow Chart

```
/*
          Start
            |
        Enter year
            |
      Is year % 4 == 0?
            |
      Yes / No
          /   \
         |     |
        Yes   No
         |     |
   Is year % 100 == 0?
         |
   Yes / No
       /   \
      |     |
      Yes   No
      |    |
  Is year % 400 == 0?
      |
 Yes  /   No
      |    |
   True  False
      |
     Output result
            |
          End
*/
```

-Explanation

A year is determined to be a leap year based on the following rules:
It is divisible by 4.
If it is divisible by 100, it must also be divisible by 400 to be considered a leap year.
The program checks these conditions using simple modulo operations and returns true or false based on the results.

-Output



-Time and Space complexity

Time Complexity:
The time complexity is **O(1)** since the program consists of a constant number of operations, regardless of the input year.

Space Complexity:
The space complexity is **O(1)** as well, as only a few variables are used to hold the input year and the result.