**C-DAC Mumbai Date25/09/2024**

**Subject: Algorithm and Data Structure**

**Assignment 1**

**Solve the assignment with 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

**Program:**

import java.util.\*;

public class Ass1prg1{

static int arm(int n)

{

if(n<10)

return n\*n\*n;

return(n%10)\*(n%10)\*(n%10)+arm(n/10);

}

public static void main(String[] args){

int num,r;

Scanner sc= new Scanner(System.in);

System.out.println("enter any number");

num = sc.nextInt();

r= arm(num);

if(num ==r)

System.out.println( "True");

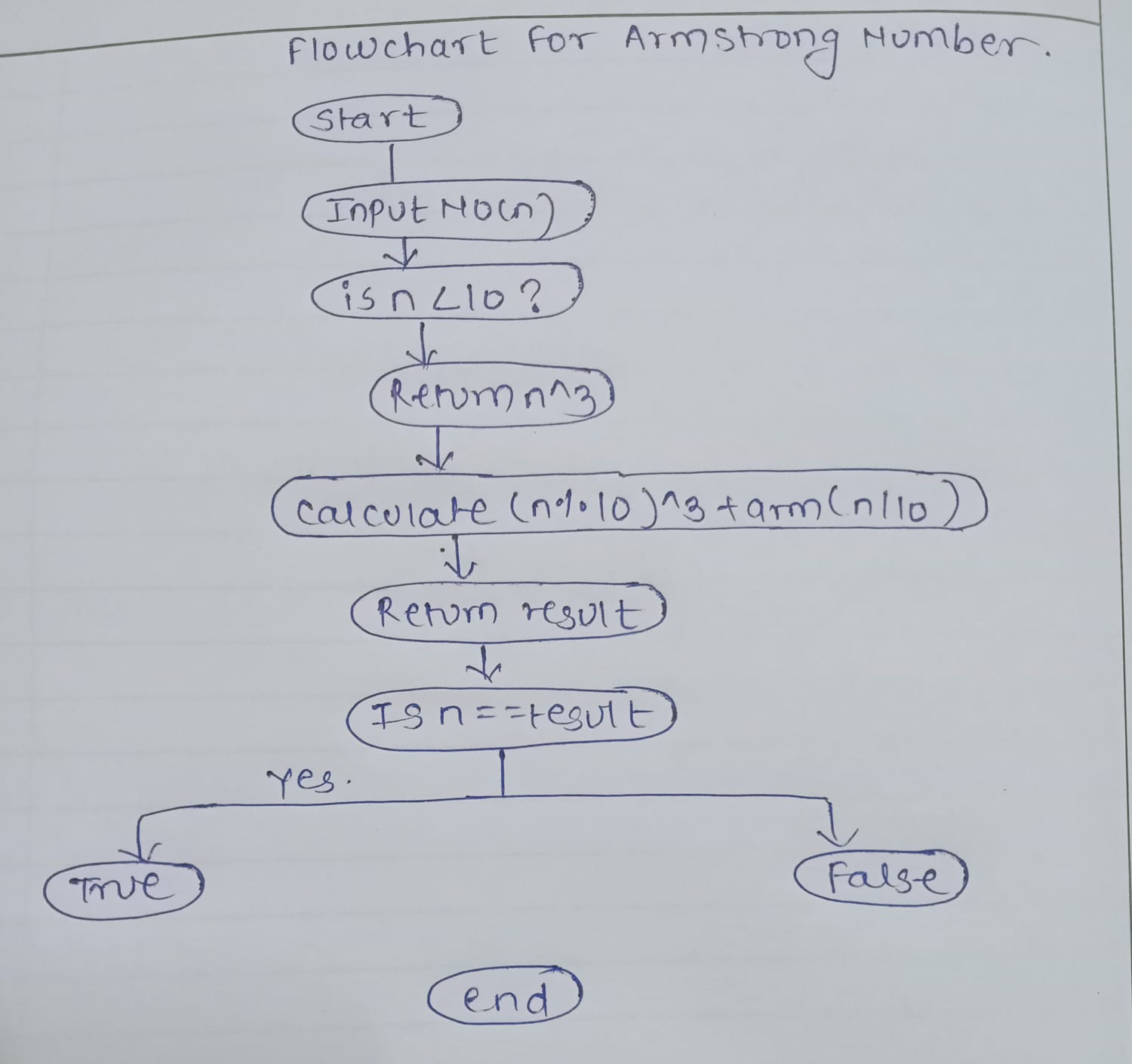
else

System.out.println( "False");

}

}

**Flowchart:**



**Explaination:**

Recursive Function arm(int n):

* This function recursively calculates the sum of the cubes of the digits of the number n.
* Base Case: When n becomes a single digit (less than 10), the function returns the cube of the single digit (n \* n \* n).
* Recursive Case: The function extracts the last digit of the number using n % 10, cubes it, and then adds this to the result of the recursive call arm(n / 10), where n / 10 removes the last digit.

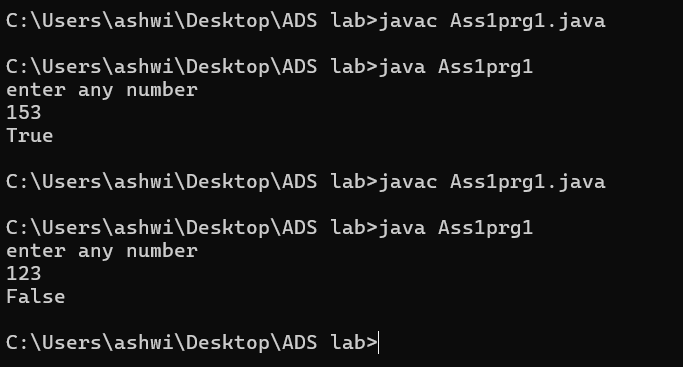
For example, for 153, the breakdown is:

153 % 10 = 3, so we calculate 3^3 = 27, then call arm(15).

15 % 10 = 5, so we calculate 5^3 = 125, then call arm(1).

1 is a single digit, so we calculate 1^3 = 1.

Now, the sum is 27+125+1 =153, which equals the original number.



**Complexity:**

Time Complexity**:** O(log​n)

Space Complexity**:** O(logn)

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

**Program:**

import java.util.\*;

public class Ass1prg2{

//recursive method

static boolean isprime(int n, int i){

//base class

if(n<=2){

return n ==2;

}

if(n%i ==0){

return false;

}

if(i\*i >n){

return true;

}

return isprime(n, i+1);

}

public static void main(String args[]){

Scanner sc = new Scanner(System.in);

System.out.println("enter a number");

int num = sc.nextInt();

if(isprime(num,2)){

System.out.println("true");

}

else{

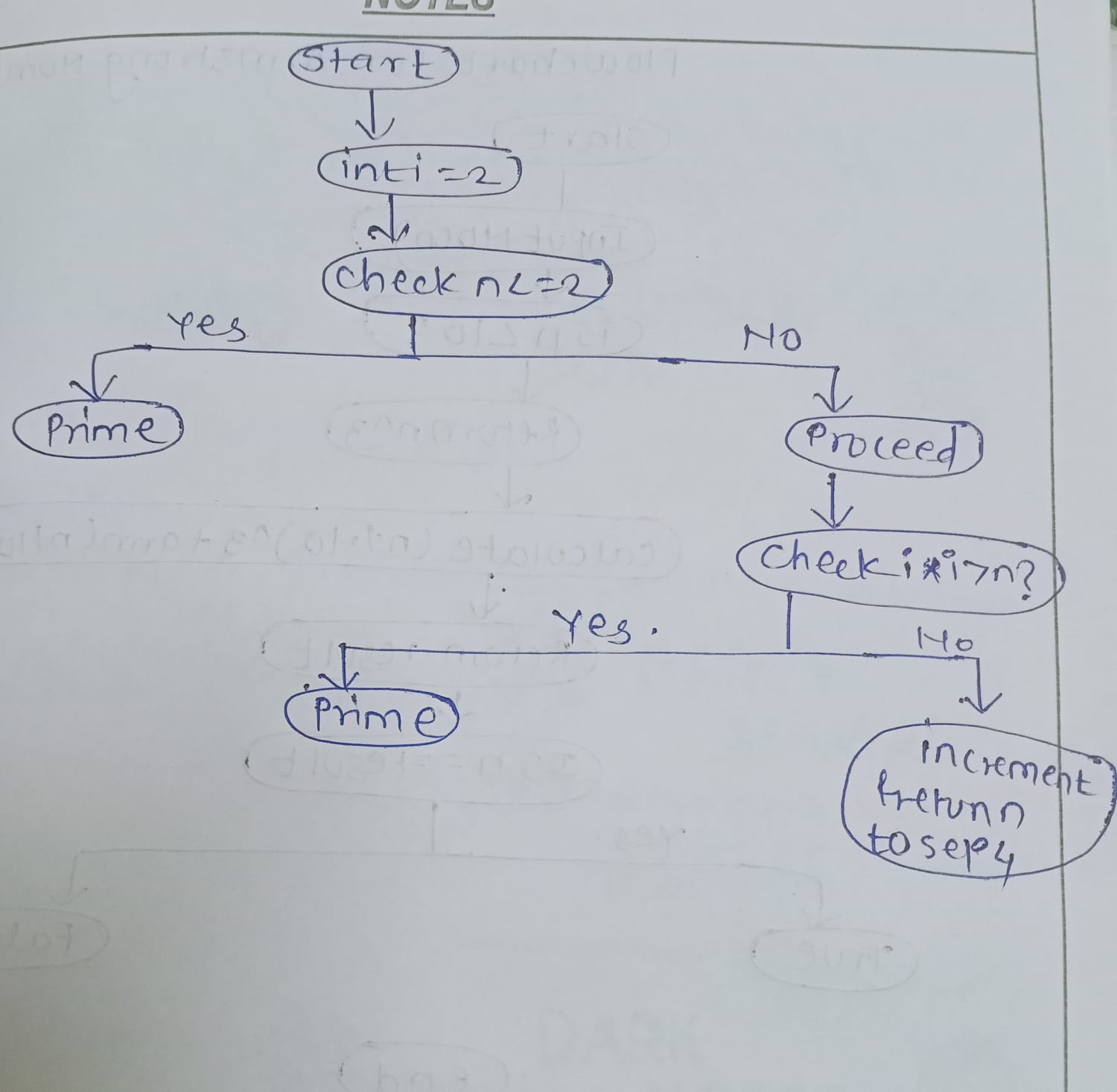
System.out.println("false");

}

}

}

**Flowchart:**



**Explanation:**

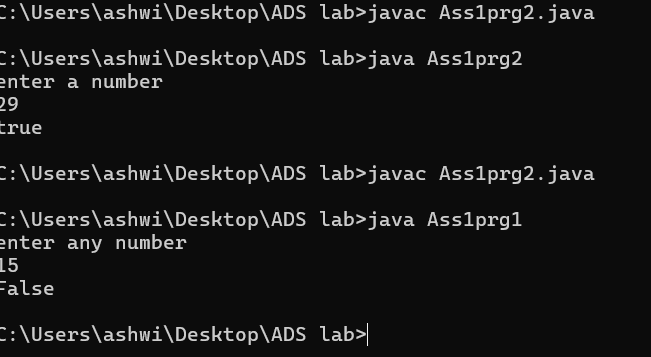
Base Cases:

* If the number n is less than or equal to 2, we return true if n is 2 .since 2 is the smallest prime number and false otherwise.
* If n % i == 0, it means n is divisible by i, so it's not a prime number.
* If i \* i > n, this means no divisor has been found up to the square root of n, so we conclude that n is prime.

Recursive Case:

* We check divisibility by the next integer i + 1 and continue this until either a divisor is found or i \* i > n which implies that no divisor exists

**Output:**



**Complexity:**

Time complexity**:** O(√n)

Space complexity**:** O(√n)

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

**Program:**

import java.util.\*;

public class Ass1prg3{

static int fact(int n){

if(n<=1){

return 1;

}

else {

return n\*fact(n-1);

}

}

public static void main(String args[]){

Scanner sc = new Scanner(System.in);

System.out.println("enter any number");

int n = sc.nextInt();

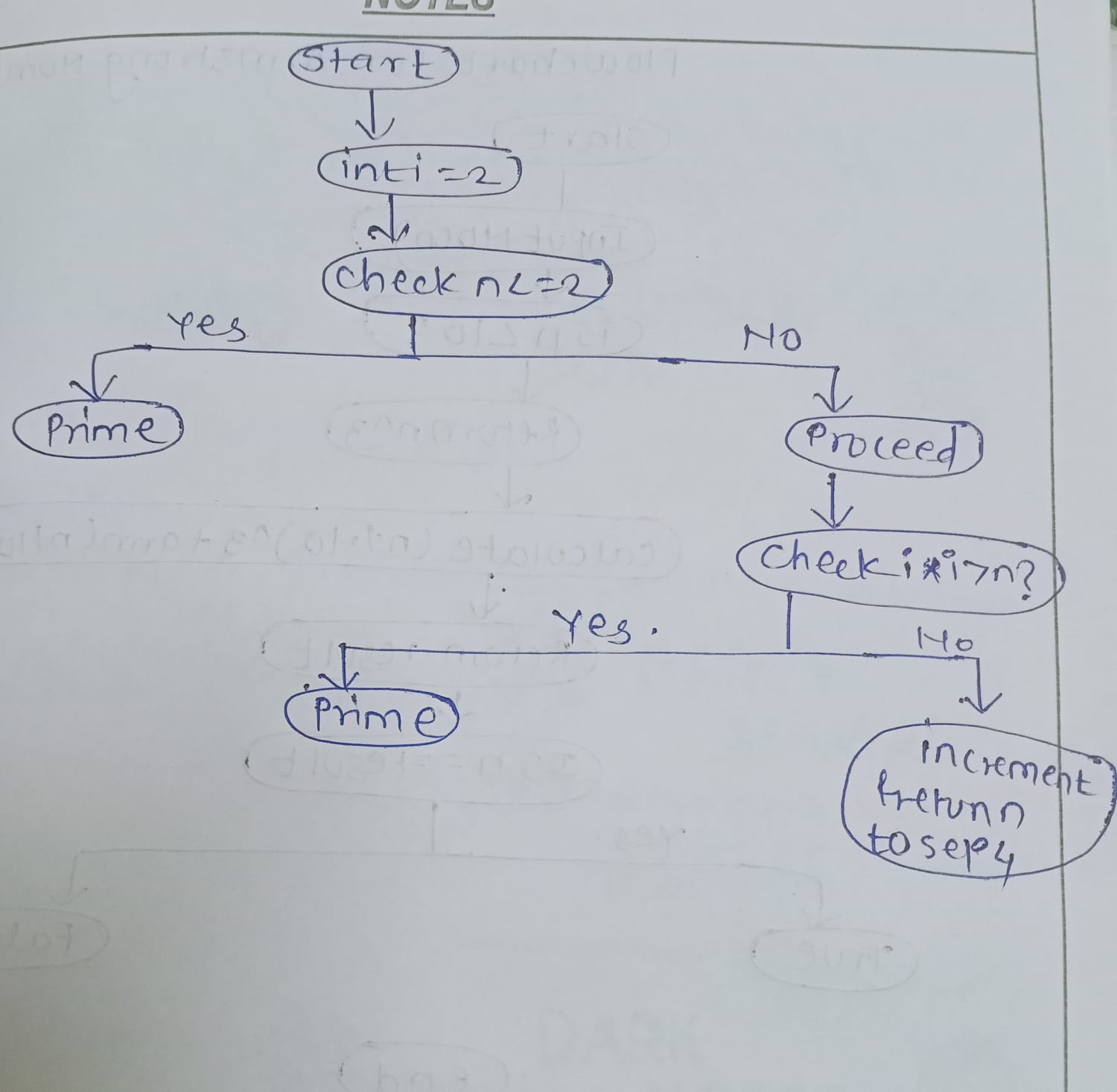
System.out.println(fact(n));

sc.close(); // Close the scanner

}

}

**Flowchart:**



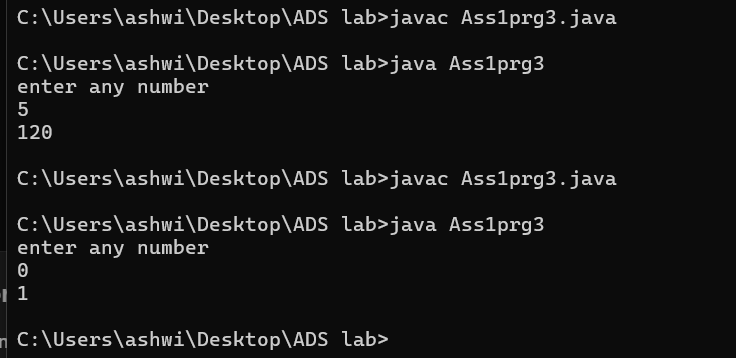
**Explanation:**

Method Signature: The method fact is static (it belongs to the class and not objects of the class), returns an int, and takes one parameter n of type int.

Base Case:

* If n is less than or equal to 1 (i.e., n == 1 or n == 0), the method returns 1. This is the base case for recursion because the factorial of 0 or 1 is defined as 1.
* If n is greater than 1, the method recursively calls itself with n - 1 and multiplies it by n. This continues until the base case is reached.

Output:



**Complexity:**

Time Complexity**:** O(n)

Space Complexity**:** O(n)

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]

**Program:**

import java.util.\*;

public class Ass1prg4{

static int fib(int n){

if(n<=1)

{

return n;

}

return fib(n-1)+fib(n-2);

}

public static void main(String args[]){

Scanner sc = new Scanner(System.in);

System.out.println("enter the number");

int n= sc.nextInt();

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

{

System.out.print(fib(i) +" ");

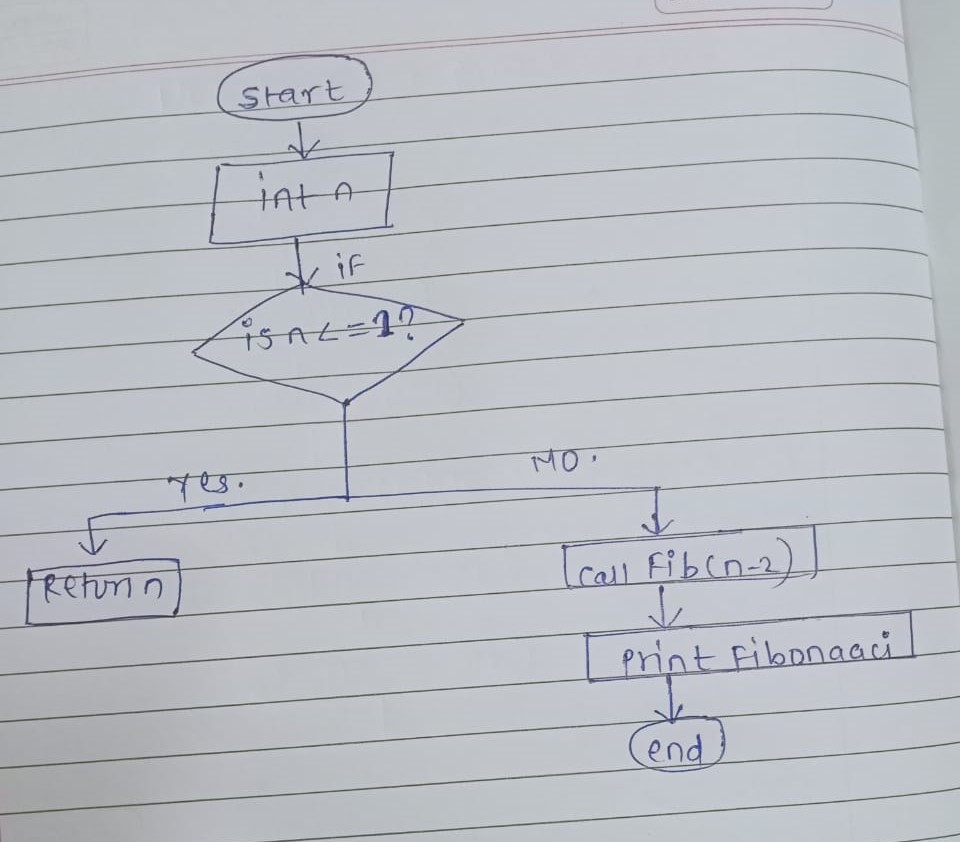
}

sc.close();

}

}

**Flowchart:**



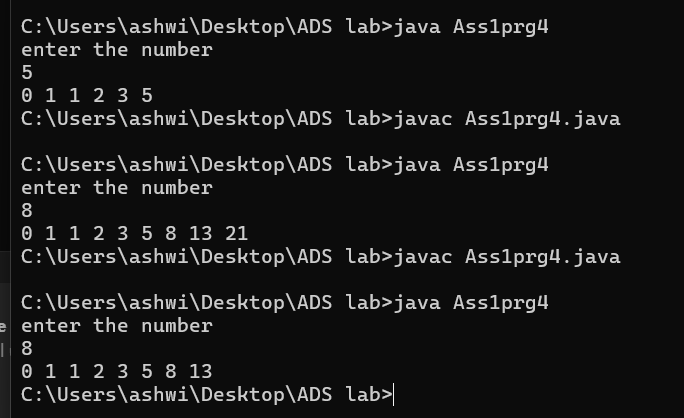
**Explanation:**

Class Declaration:

* public class Ass1prg4: This declares a public class named Ass1prg4.

Fibonacci Method:

* static int fib(int n): This method computes the Fibonacci number at position n using recursion.
  + Base Case:
    - if (n <= 1) return n;: If n is 0 or 1, it returns n, effectively defining fib(0) = 0 and fib(1) = 1.
  + Recursive Case:
    - return fib(n - 1) + fib(n - 2);: For any other value of n, the method recursively calls itself to compute the sum of the two preceding Fibonacci numbers.



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

**Program:**

import java.util.\*;

public class Ass1prg5{

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.println("enter any number");

int a= sc.nextInt();

int b= sc.nextInt();

int n =findGCD(a,b);

System.out.println("GCD =" +n);

sc.close();

}

private static int findGCD(int a, int b){

if(b ==0)

return a;

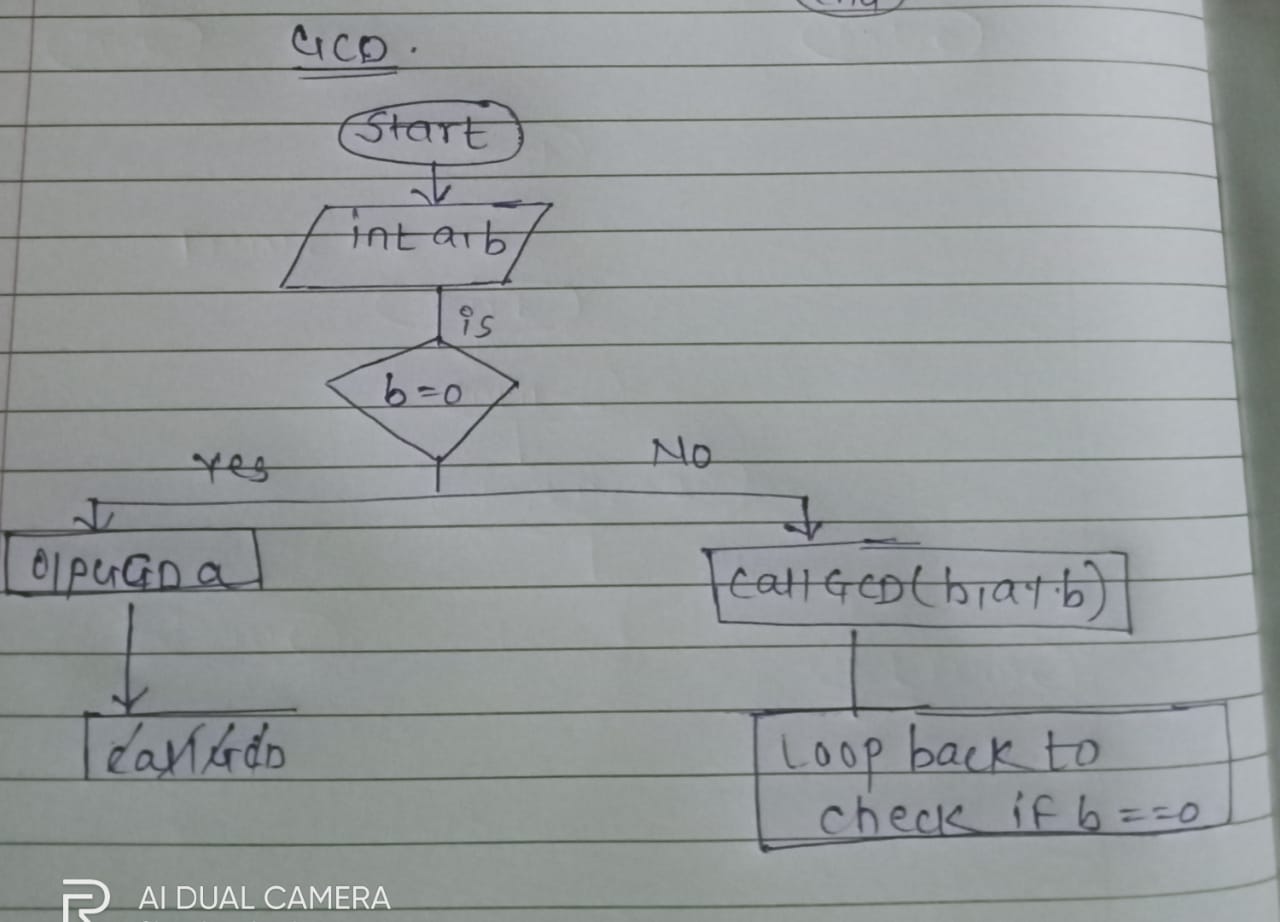
else

return findGCD(b,a%b);

}

}

**Flowchart:**



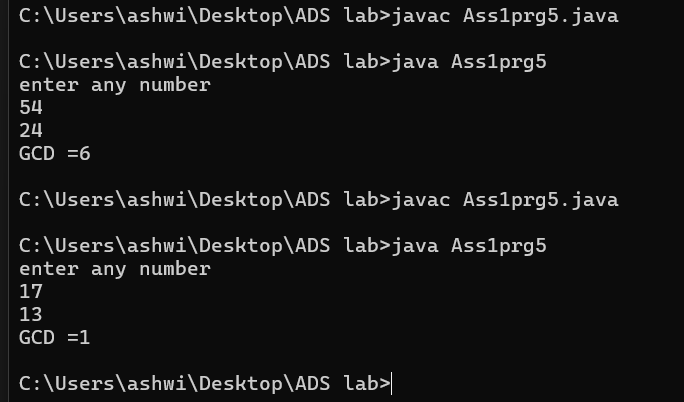
**Explanation:**

1. GCD Calculation:
   * The findGCD method uses the Euclidean algorithm:
     + If the second number (b) is zero, it returns the first number (a) as the GCD.
     + Otherwise, it calls itself with the parameters (b, a % b).
2. Output: The program prints the GCD of the two numbers.

Example

* Input: 54 and 24
* Output: GCD = 6

**Output:**



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

**Program:**

public class Ass1prg6{

public static int findSqrt(int x, int i) {

if (i \* i > x) {

return i - 1;

}

return findSqrt(x, i + 1);

}

public static void main(String[] args) {

int x1 = 16;

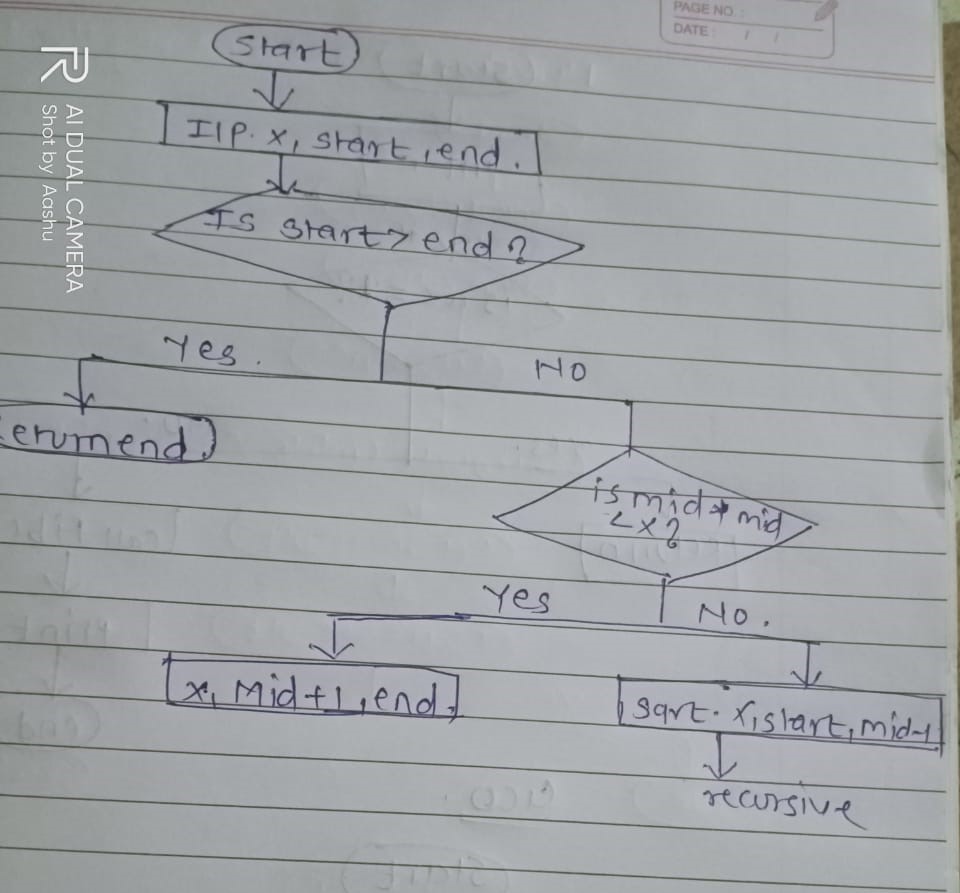
int x2 = 27;

System.out.println("Square root of " + x1 + " is: " + findSqrt(x1, 1));

System.out.println("Square root of " + x2 + " is: " + findSqrt(x2, 1));

}

}

**Flowchart: **

**Explanation:**

The program takes an integer x and recursively finds its integer square root.

The recursive function findSqrt(x, i) starts checking from i = 1 and increases i until i \* i exceeds x.

Once i \* i exceeds x, it returns i - 1 as the integer square root because i \* i is the first number that exceeds x.

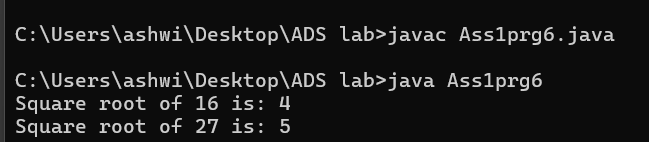
The function terminates once the correct integer square root is found.

**Complexity:**

Time Complexity: O(√n)

Space complexity: O(√n)

**Output:**



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']

**Program:**

public class Ass1prg7 {

public static void findRepeatedCharacters(String str) {

int[] charCount = new int[256];

for (int i = 0; i < str.length(); i++) {

charCount[str.charAt(i)]++;

}

System.out.print("Repeated characters: ");

for (int i = 0; i < 256; i++) {

if (charCount[i] > 1) {

System.out.print((char) i + " ");

}

}

System.out.println();

}

public static void main(String[] args) {

String input1 = "programming";

System.out.println("Input: " + input1);

findRepeatedCharacters(input1); // Expected Output: r g m

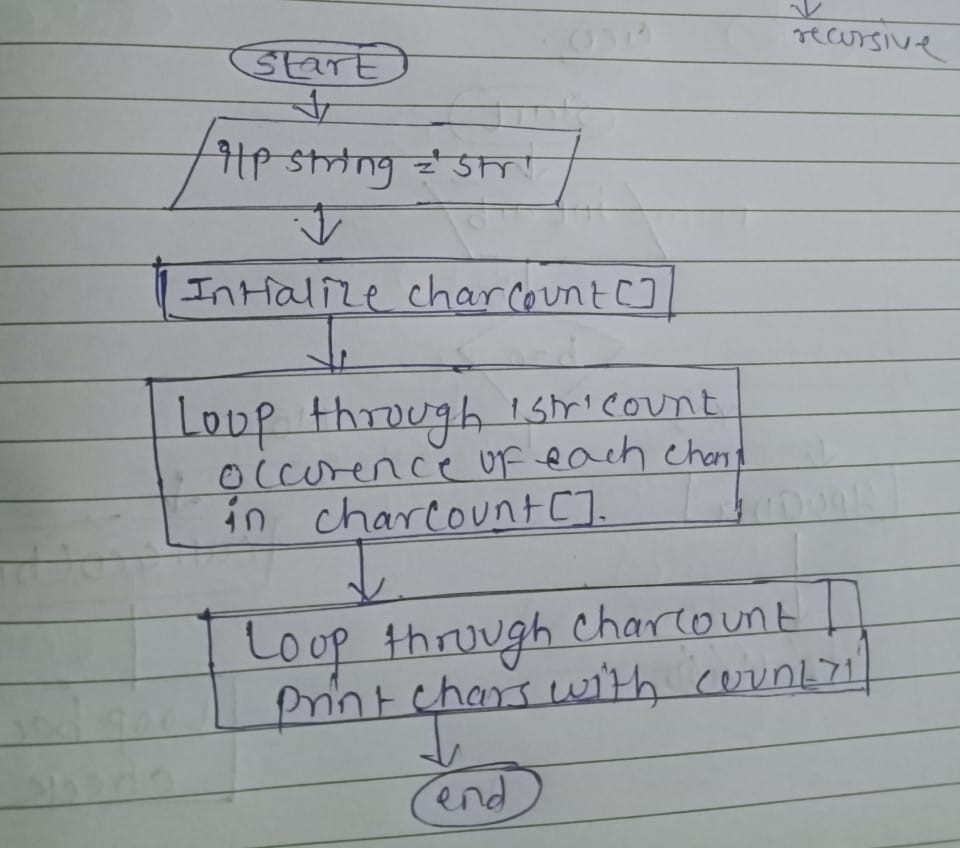
String input2 = "hello";

System.out.println("Input: " + input2);

findRepeatedCharacters(input2); // Expected Output: l

}

}

**Flowchart:** 

**Explanation:**

Start

Input String str

Initialize an Array charCount of size 256 to count the occurrence of each character.

Loop through each character in the string str:

* For each character str[i], increment its count in the charCount array.

Loop through the charCount array:

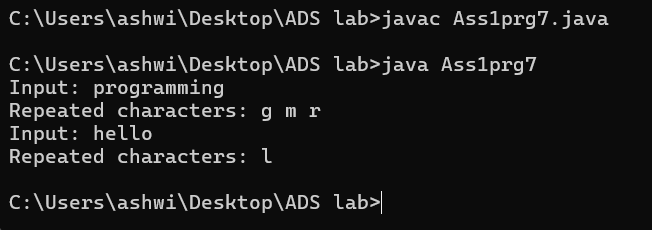
* If any character's count is greater than 1, it is repeated, so print that character.

End

**Complexity:**

Time Complexity:O[n]

spaceComplexity:O[n]



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

**Program:**

public class Ass1prg8{

public static char findFirstNonRepeatedCharacter(String str) {

int[] charCount = new int[256];

for (int i = 0; i < str.length(); i++) {

charCount[str.charAt(i)]++;

}

for (int i = 0; i < str.length(); i++) {

if (charCount[str.charAt(i)] == 1) {

return str.charAt(i); // Return the first non-repeated character

}

}

return '-';

}

public static void main(String[] args) {

String input1 = "stress";

System.out.println("Input: " + input1);

char result1 = findFirstNonRepeatedCharacter(input1);

System.out.println("First non-repeated character: " + (result1 != '-' ? result1 : "null"));

String input2 = "aabbcc";

System.out.println("Input: " + input2);

char result2 = findFirstNonRepeatedCharacter(input2);

System.out.println("First non-repeated character: " + (result2 != '-' ? result2 : "null"));

}

}

**Explanation:**

Character Count Array:

We use an array charCount of size 256 (for ASCII characters) to store how many times each character appears in the string.

First Loop:

We loop through the string and count the occurrences of each character by using the character's ASCII value as the index in the array.

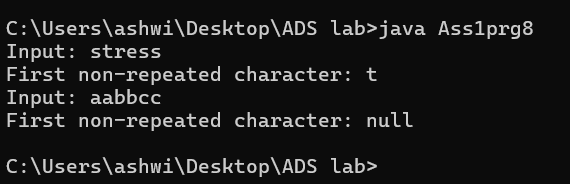
Second Loop:

We then loop through the string again to check which character has a count of 1. The first character that meets this condition is the first non-repeated character.

Return Value:

If we find a non-repeated character, we return it. If no non-repeated character is found, we return a special character like '-', which indicates that no such character exists we print "null" in this case

**Output:**



**Complexity:**

Time Complexity: O(n)

Space Complexity:O(1)

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

**Program:**

public class Ass1prg9 {

public static boolean isPalindrome(int num) {

if (num < 0) {

return false;

}

int originalNum = num;

int reversedNum = 0;

while (num != 0) {

int lastDigit = num % 10;

reversedNum = reversedNum \* 10 + lastDigit;

num = num / 10;

}

return originalNum == reversedNum;

}

public static void main(String[] args) {

int input1 = 121;

System.out.println("Input: " + input1);

System.out.println("Is Palindrome: " + isPalindrome(input1));

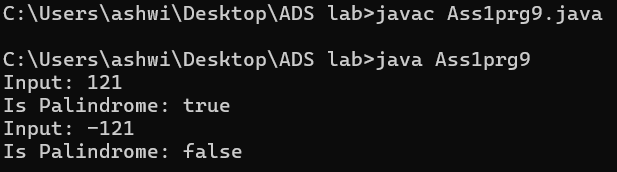
int input2 = -121;

System.out.println("Input: " + input2);

System.out.println("Is Palindrome: " + isPalindrome(input2));

}

}

**Output:** 

**Complexity:**

Time complexity:O(log₁₀(n)):

space complexity:O(l):

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

**Program:**

**public class Ass1prg10 {**

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) {

int year1 = 2020;

System.out.println("Year: " + year1);

System.out.println("Is Leap Year: " + isLeapYear(year1)); // Expected Output: true

int year2 = 1900;

System.out.println("Year: " + year2);

System.out.println("Is Leap Year: " + isLeapYear(year2)); // Expected Output: false

}

}

**Complexity:**

Time Complexity: O(1)

space Complexity: O(1)

**Output:**

