1. **Armstrong Number**

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

import java.util.Scanner;

public class Armstrong{

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.print("Enter a number: ");

int n = sc.nextInt(); //153

int nBackup = n; //take backup of n

int count = Integer.toString(n).length();

int sum=0, lastDigit;

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

lastDigit = n%10; //3 ; 5

sum = sum + (int)Math.pow(lastDigit, count); // 0 + 3^3 ; 3^3 + 5^3

n /= 10; //15 ;

}

if(sum == nBackup){

System.out.println("true");

}

else

System.out.println("false");

}

}

/\* Armstrong No.-> 153 = 1^3 + 5^3 + 3^3 (No. of digits is 3)

1. Take input from user

2. Calculate the count of digits in the input number eg. 153 -> 3 digits (convert int to String then find length of string)

int count = Integer.toString(n).length();

3. Now, we need to calculate 1^3 + 5^3 + 3^3 and then compare it with original value

4. Using for loop, separate the input number into individual digits using n%10 eg. 153%10 = 3; 15%10 = 5

5. Add the digit^count (3^3;5^3;1^3) into sum --> sum = sum + (int)Math.pow(lastDigit, count);

6. Keep dividing n/10 so that we can remove the units place

7. If sum == input number (3^3+5^3+1^3 == 153) -> Then it is Armstrong

Time Complexity -> O(n)

Space Complexity -> O(1)

\*/

//Using Recursion

import java.util.Scanner;

public class ArmstrongRec{

public static int checkArmstrong(int n, int count, int sum){

if(n == 0)

return sum;

int lastDigit = n%10;

sum = sum + (int)Math.pow(lastDigit, count);

return checkArmstrong(n/10, count, sum);

}

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.print("Enter a number: ");

int n = sc.nextInt();

int nBackup = n;

int count = Integer.toString(n).length();

int sum = 0;

sum = checkArmstrong(n, count, sum);

if(sum == nBackup){

System.out.println("true");

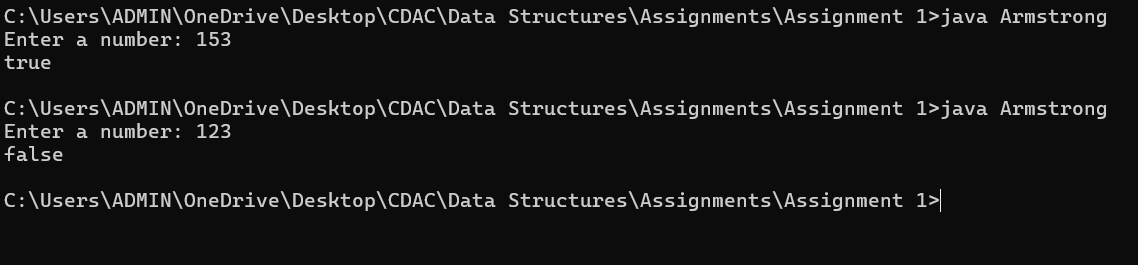
}

else

System.out.println("false");

}

}



2. **Prime Number**

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

import java.util.Scanner;

public class PrimeNumber{

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.print("Enter a number: ");

int n = sc.nextInt();

int count = 0;

for(int i=1; i<=n; i++){

if(n%i==0){ //Check if the input number is completely divisible by any number from 1 to itself

count++; // Increment count when the input number is completely divisble by any number from 1 to itself

}

}

if(count == 2) //Prime number should only have 2 cofactors i.e. 1 and itself. Therefore, if count == 2, then it is a prime number

System.out.println("true");

else

System.out.println("false");

}

}

//Time Complexity -> O(n)

//Space Complexity -> O(1)

//Using Recursion

import java.util.Scanner;

public class PrimeNumberRec{

public static boolean checkPrime(int n, int count, int i){

if(i>n){

if(count==2) return true;

return false;

}

if(n%i == 0) count++;

return checkPrime(n,count,i+1);

}

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.print("Enter a number: ");

int n = sc.nextInt();

int count = 0, i=1;

boolean primeStatus = checkPrime(n, count, i);

if(primeStatus == true)

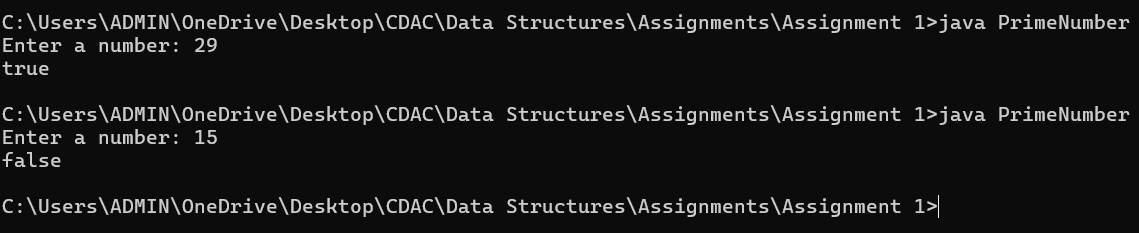
System.out.println("true");

else

System.out.println("false");

}

}



3. **Factorial**

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

//Uses recursion

import java.util.Scanner;

// Factorial -> n! = n \* (n-1)! eg. 5! = 5 \* 4 \* 3 \* 2 \* 1 -> 5 \* 4!

//4! = 4 \* 3 \* 2 \* 1 -> 4 \* 3!

//So if 5! is f(5) then f(5) = 5 \* f(5-1) -> f(n) = n \* f(n-1)

public class Factorial{

public static int calcFactorial(int n){

if(n==1 || n==0){ // 1! and 0! is 1. There return 1 for f(1) and f(0)

return 1;

}

int factorial = n \* calcFactorial(n-1); // f(n) = n \* f(n-1) -> Recursive call

return factorial;

}

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.print("Enter a number: ");

int n = sc.nextInt();

int factorial = calcFactorial(n); // call method which will return the factorial value

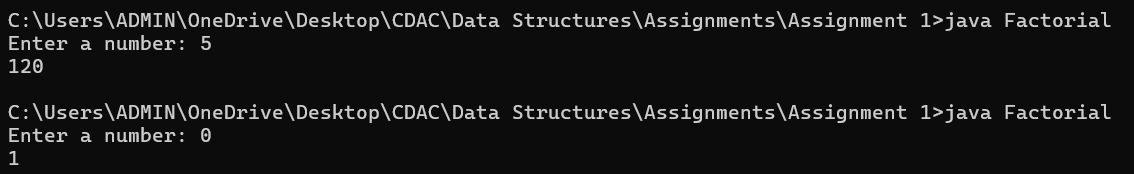
System.out.println(factorial);

}

}

//Time Complexity -> O(n) -> Since the n will keep decrementing until 1 is reached in each call, so n recursive calls are made

//Space Complexity -> O(n) -> as new stack frame per call will get space in memory n times



4. **Fibonacci Series**

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

import java.util.Scanner;

import java.util.Arrays;

public class Fibonacci{

public static int calcFib(int n){

if(n==0 || n==1){ // Fibonacci till 0th index is 0 and till 1st index is 1.

return n; //Therefore, return 0 or 1 if n is 0 or 1 respectively.

}

int fib = calcFib(n-1) + calcFib(n-2); //if current element is f(n) then f(n) = f(n-1) + f(n-2). Therefore, call the function recursively for prev elements

return fib;

}

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.print("Enter a number: "); // n = 5 0 1 1 2 3

int n = sc.nextInt();

int[] arr = new int[n];

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

arr[i] = calcFib(i); // call the method to calculate fibonacci number at each index and store in array

System.out.print(Arrays.toString(arr));

}

}

//Height of this binary tree -> 5 i.e n

// 5 level0 max nodes-> 2^0 1

// 4 3 level1 max nodes-> 2^1 2

// 3 2 2 1 level2 max nodes-> 2^2 4

// 2 1 1 0 1 0 level3 max nodes-> 2^3 8

//1 0 level4 max nodes-> 2^4 16

// Here, nodes are the recursive calls

// Max no. of nodes in a binary tree = 2^n = 2^5 = 32

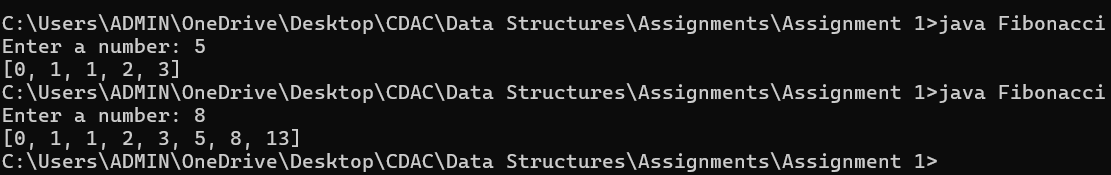
//Therefore, Time Complexity -> O(2^n)

//Space Complexity -> O(n)

//Since at any point, max no. of stacks created will be equal to n as they get destroyed every time they return control to their calling stack

// E.g. 5 -> 4 and 3 but f(4) will get called first then -> 3 and 2 but f(3) will get called first and so on

// so it will be f(5) -> f(4) ->f(3) ->f(2) ->f(1) -> Will return control back



5. **Find GCD**

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

import java.util.Scanner;

public class GCD{

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.print("Enter a: ");

int a = sc.nextInt();

System.out.print("Enter b: ");

int b = sc.nextInt();

// 5 , 15

// a b

// b>a -> b%a, a -> 15%5 = 0, 5 -> if any number becomes 0, the other number is the GCD (Euclideans Algorithm)

// a>b -> a%b, b

while(a !=0 && b!=0)

{

if(a>b)

a = a%b;

else

b = b%a;

}

if(a==0)

System.out.print(b);

if(b==0)

System.out.print(a);

}

}

//Time Complexity -> O(log n) -> The numbers are getting divided in each loop

//Space Complexity -> O(1)

//Using recursion

import java.util.Scanner;

public class GCDRec{

public static int calcGCD(int a, int b){

if(a==0) return b; //if any number is 0, then the other remaining value should be the GCD

if(b==0) return a;

if(a>b) a = a%b;

else b = b%a;

return calcGCD(a, b); //pass the modified values of a and b to calcGCD(int a, int b)

}

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.print("Enter a: ");

int a = sc.nextInt();

System.out.print("Enter b: ");

int b = sc.nextInt();

// 5 , 15

// a b

// b>a -> b%a, a -> 15%5 = 0, 5 -> if any number becomes 0, the other number is the GCD (Euclideans Algorithm)

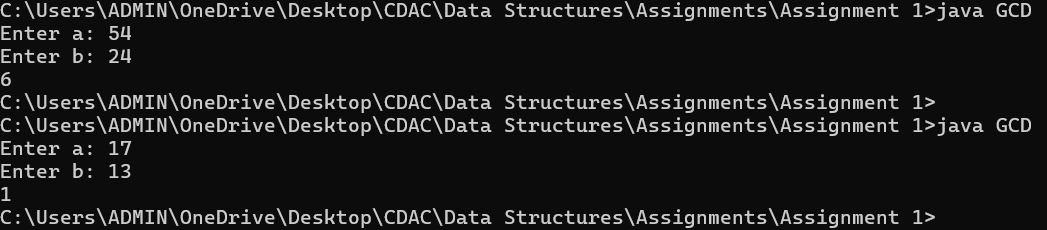
// a>b -> a%b, b

int gcd = calcGCD(a, b); //call calcGCD(int a, int b) which will return the GCD value

System.out.print("GCD is " + gcd);

}

}



6. **Find Square Root**

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

import java.util.Scanner;

public class SquareRoot{

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.print("Enter a number: ");

int n = sc.nextInt();

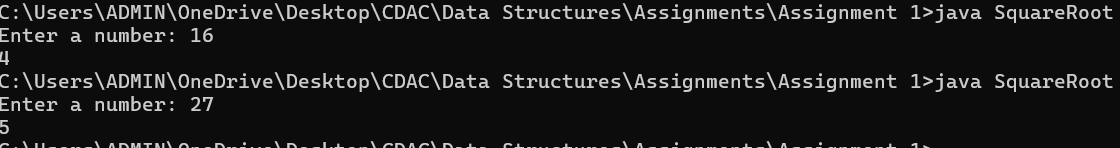
int squareRoot = (int)Math.sqrt(n); // cast result of Math.sqrt to integer

System.out.print(squareRoot);

}}

//Time Complexity -> O(1)

//Space Complexity -> O(1)



7. **Find Repeated Characters in a String**

Problem: Write a Java program to find all repeated characters in a string.

import java.util.Scanner;

import java.util.\*;

public class RepeatedCharacters{

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.print("Enter a string: ");

String str = sc.nextLine(); //input a string

Set<Character> chSet = new HashSet<>(); // create a set to store the unique repeated characters

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

// check if a character in the string is present at more than one index. If there is only occurence of the char, then its first occurence index should be equal to its last occurence index E.g. -> gaming -> indexes of g are 0 and 5 thus it has repeated

// a's first and last index is same that is 1. Thus, it is not repeated in the String

if( (str.indexOf(str.charAt(i))) != (str.lastIndexOf(str.charAt(i))) ) //checking if indexes of first and last occurence of char in the string are not equal

chSet.add(str.charAt(i)); // add the repeated characters in string to Set since we want the unique values from all the repeated letters

}

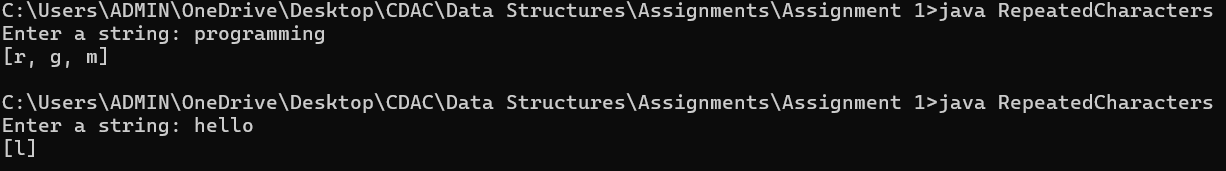
System.out.println(chSet);

}

}

//Time Complexity -> O(n^2) -> In for loop, indexOf and lastIndexOf also use loops to check for indexes

//Space Complexity -> O(n)



8. **First Non-Repeated Character**

Problem: Write a Java program to find the first non-repeated character in a string.’

import java.util.Scanner;

import java.util.\*;

public class FirstNonRepeatedCharacter{

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.print("Enter a string: ");

String str = sc.nextLine();

boolean charFound = false;

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

if( (str.indexOf(str.charAt(i))) == (str.lastIndexOf(str.charAt(i))) ){ // The unique character which has no duplicates must have same first and last index

System.out.println("'" + str.charAt(i) + "'");

charFound = true; // Break out of loop as soon as first unique character is found

break;

}

}

if (charFound == false)

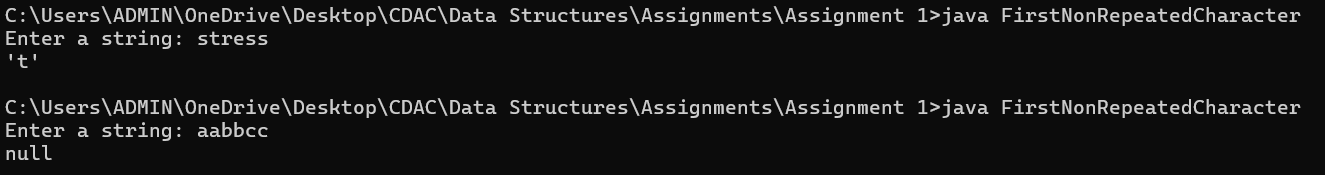
System.out.println("null");

}

}

//Time Complexity -> O(n^2) -> In for loop, indexOf and lastIndexOf also use loops to check for indexes

//Space Complexity -> O(n)



9. **Integer Palindrome**

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

import java.util.Scanner;

import java.util.\*;

public class Palindrome{

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

// System.out.print("Enter a string: ");

String str = sc.nextLine();

StringBuilder reverseStr = new StringBuilder();

//for eg. consider String -> 121

for(int i=str.length() - 1; i>=0; i--){ // traverse from last index(str.length() - 1) to 0 index

reverseStr.append(str.charAt(i)); // append letters from last to first in another string (StringBuilder to append)

}

if(str.compareTo(reverseStr.toString()) == 0){ // compare the original string with reversed string (compareTo returns 0 if they are equal)

System.out.print("true");

}

else

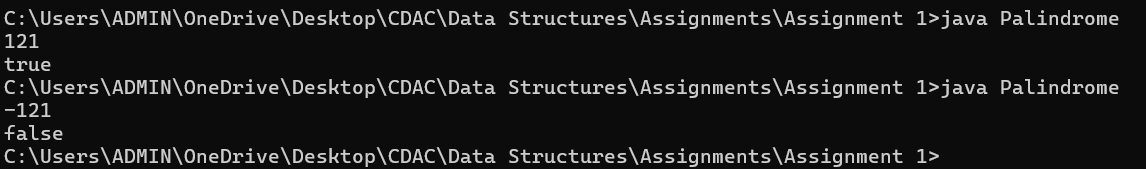
System.out.print("false");

}

}

//Time Complexity -> O(n)

//Space Complexity -> O(n)



10. **Leap Year**

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

import java.util.Scanner;

public class LeapYear{

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.print("Enter year: ");

int year = sc.nextInt();

if( (year % 4 == 0 && year % 100 != 0 ) || year % 400 ==0) // Condition to check leap year (divisible by 4 and not divisible by 100 OR divisible by 400)

System.out.print("true");

else

System.out.print("false");

}

}

//Time Complexity -> O(1)

//Space Complexity -> O(1)

