**C-DAC Mumbai Date 25/09/2024**

**Subject: Algorithm and Data Structure**

**Assignment 1**

**1. Armstrong Number:**

**Program:**

Java

import java.util.Scanner;

public class ArmstrongNumber {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

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

int number = scanner.nextInt();

int originalNumber = number;

int sumOfPowers

= 0;

int numberOfDigits = (int) Math.log10(number) + 1;

while (number != 0) {

int digit = number % 10;

sumOfPowers += Math.pow(digit, numberOfDigits);

number /= 10;

}

if (sumOfPowers == originalNumber) {

System.out.println(originalNumber + " is an Armstrong number.");

} else {

System.out.println(originalNumber + " is not an Armstrong number.");

}

}

}

**Explanation:**

1. The program takes a number as input.
2. It calculates the number of digits in the input number.
3. It iterates through each digit of the number:
   * Calculates the power of the digit raised to the number of digits.
   * Adds the result to the sum of powers.
4. If the sum of powers equals the original number, it's an Armstrong number; otherwise, it's not.

**Output:**

Enter a number: 153

153 is an Armstrong number.

**Time and Space Complexity:**

* Time complexity: O(log n), where n is the input number.
* Space complexity: O(1), as the program uses constant space for variables.

**2. Prime Number:**

**Program:**

Java

import java.util.Scanner;

public class PrimeNumber {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

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

int number = scanner.nextInt();

if (number <= 1) {

System.out.println(number

+ " is not a prime number.");

return;

}

boolean isPrime = true;

for (int i = 2; i <= Math.sqrt(number); i++) {

if (number % i == 0) {

isPrime = false;

break;

}

}

if (isPrime) {

System.out.println(number

+ " is a prime number.");

} else {

System.out.println(number + " is not a prime number.");

}

}

}

**Explanation:**

1. The program takes a number as input.
2. It checks if the number is less than or equal to 1 (not prime).
3. It iterates from 2 to the square root of the number:
   * If the number is divisible by any of these values, it's not prime.
4. If the number is not divisible by any value from 2 to its square root, it's prime.

**Output:**

Enter a number: 29

29 is a prime number.

**Time and Space Complexity:**

* Time complexity: O(sqrt(n)), where n is the input number.
* Space complexity: O(1), as the program uses constant space for variables.

**3. Factorial:**

**Program:**

Java

import java.util.Scanner;

public class Factorial {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter

a number: ");

int number = scanner.nextInt();

if (number < 0) {

System.out.println("Factorial is not defined for negative numbers.");

return;

}

int factorial = 1;

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

factorial \*= i;

}

System.out.println("Factorial of " + number + " is " + factorial);

}

}

**Explanation:**

1. The program takes a number as input.
2. It checks if the number is negative (factorial not defined).
3. It iterates from 1 to the input number, multiplying each value with the factorial.
4. It prints the calculated factorial.

**Output:**

Enter a number: 5

Factorial of 5 is 120

**Time and Space Complexity:**

* Time complexity: O(n), where n is the input number.
* Space complexity: O(1), as the program uses constant space for variables.

**4. Fibonacci Series:**

**Program:**

Java

import java.util.Scanner;

import java.util.ArrayList;

import java.util.List;

public class FibonacciSeries {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter

the number of terms: ");

int n = scanner.nextInt();

if (n <= 0) {

System.out.println("Please

enter a positive number.");

return;

}

List<Integer> fibonacciSeries = new ArrayList<>();

fibonacciSeries.add(0);

fibonacciSeries.add(1);

for (int i = 2; i < n; i++) {

int nextTerm = fibonacciSeries.get(i - 1) + fibonacciSeries.get(i - 2);

fibonacciSeries.add(nextTerm);

}

System.out.println("Fibonacci series: " + fibonacciSeries);

}

}

**Explanation:**

1. The program takes the number of terms as input.
2. It checks if the number is positive.
3. It initializes the Fibonacci series with the first two terms (0 and 1).
4. It iterates from 2 to the number of terms:
   * Calculates the next term by adding the previous two terms.
   * Adds the next term to the series.
5. It prints the generated Fibonacci series.

**Output:**

Enter the number of terms: 5

Fibonacci series: [0, 1, 1, 2, 3]

**Time and Space Complexity:**

* Time complexity: O(n), where n is the number of terms.
* Space complexity: O(n), as the program stores the Fibonacci series in a list.

**5. Find GCD:**

**Program:**

Java

import java.util.Scanner;

public class GCD {

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

if (b == 0) {

return a;

} else {

return gcd(b, a % b);

}

}

public static void main(String[] args) {

Scanner

scanner = new Scanner(System.in);

System.out.print("Enter the first number: ");

int a = scanner.nextInt();

System.out.print("Enter the second number: ");

int b = scanner.nextInt();

int result = gcd(a, b);

System.out.println("GCD of " + a + " and " + b + " is " + result);

}

}

**Explanation:**

1. The program takes two numbers as input.
2. It calls the gcd function to recursively calculate the GCD.
3. The gcd function uses the Euclidean algorithm:
   * If the second number is 0, the GCD is the first number.
   * Otherwise, it recursively calls itself with the second number as the first and the remainder as the second.

**Output:**

Enter the first number: 54

Enter the second number: 24

GCD of 54 and 24 is 6

**Time and Space Complexity:**

* Time complexity: O(log(max(a, b))), where a and b are the input numbers.
* Space complexity: O(log(max(a, b))) due to the recursive calls.

**6. Find Square Root:**

**6. Find Square Root:**

**Program:**

Java

import java.util.Scanner;

public class SquareRoot {

public static int squareRoot(int x) {

if (x == 0 || x == 1) {

return x;

}

int i = 1;

int result = 1;

while (result <= x) {

i++;

result = i \* i;

}

return i - 1;

}

public static void main(String[] args)

{

Scanner scanner = new Scanner(System.in);

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

int number = scanner.nextInt();

int

squareRoot = squareRoot(number);

System.out.println("Square root of " + number + " is " + squareRoot);

}

}

**Explanation:**

1. The program takes a number as input.
2. It checks if the number is 0 or 1 (special cases).
3. It iterates from 1 to the square root of the input number:
   * Calculates the square of the current number.
   * If the square is greater than the input number, the square root is the previous number.
4. It returns the calculated square root.

**Output:**

Enter a number: 16

Square root of 16 is 4

**Time and Space Complexity:**

* Time complexity: O(sqrt(n)), where n is the input number.
* Space complexity: O(1), as the program uses constant space for variables.

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

**Program:**

Java

import java.util.HashMap;

import java.util.Map;

import java.util.Set;

public class RepeatedCharacters {

public static Set<Character> findRepeatedCharacters(String str) {

Map<Character, Integer> charCount = new HashMap<>();

Set<Character> repeatedCharacters = new HashSet<>();

for (char c : str.toCharArray()) {

charCount.put(c, charCount.getOrDefault(c, 0) + 1);

}

for (Map.Entry<Character, Integer> entry : charCount.entrySet()) {

if (entry.getValue()

> 1) {

repeatedCharacters.add(entry.getKey());

}

}

return repeatedCharacters;

}

public static void main(String[] args) {

String str = "programming";

Set<Character> repeatedChars = findRepeatedCharacters(str);

System.out.println("Repeated characters in " + str + ": " + repeatedChars);

}

}

**Explanation:**

1. The program takes a string as input.
2. It creates a map to store character counts.
3. It iterates through each character in the string:
   * Increments the count of the character in the map.
4. It iterates through the map entries:
   * If the count of a character is greater than 1, it's a repeated character.
5. It returns the set of repeated characters.

**Output:**

Repeated characters in programming: [r, g, m]

**Time and Space Complexity:**

* Time complexity: O(n), where n is the length of the string.
* Space complexity: O(n) for the map and set.

**8. First Non-Repeated Character:**

**Program:**

Java

import java.util.HashMap;

import java.util.Map;

public class FirstNonRepeatedCharacter {

public static Character findFirstNonRepeatedCharacter(String str) {

Map<Character, Integer> charCount = new HashMap<>();

for (char c : str.toCharArray()) {

charCount.put(c, charCount.getOrDefault(c, 0) + 1);

}

for (char c : str.toCharArray()) {

if (charCount.get(c) == 1) {

return c;

}

}

return

null;

}

public static void main(String[] args) {

String str = "stress";

Character firstNonRepeatedChar = findFirstNonRepeatedCharacter(str);

if (firstNonRepeatedChar != null) {

System.out.println("First non-repeated character in " + str + " is " + firstNonRepeatedChar);

} else {

System.out.println("No non-repeated character found in " + str);

}

}

}

**Explanation:**

1. The program takes a string as input.
2. It creates a map to store character counts.
3. It iterates through each character in the string:
   * Increments the count of the character in the map.
4. It iterates through the string again:
   * If the count of a character is 1, it's the first non-repeated character.
5. It returns the first non-repeated character or null if not found.

**Output:**

First non-repeated character in stress is t

**Time and Space Complexity:**

* Time complexity: O(n), where n is the length of the string.
* Space complexity: O(n) for the map.

**9. Integer Palindrome:**

**Program:**

Java

public class IntegerPalindrome {

public static boolean isPalindrome(int x) {

if (x < 0 || (x % 10 == 0 && x != 0)) {

return false;

}

int reversedNumber = 0;

int originalNumber = x;

while (x != 0) {

int digit = x % 10;

reversedNumber = reversedNumber \* 10 + digit;

x /= 10;

}

return

reversedNumber == originalNumber;

}

public static void main(String[] args) {

int number = 121;

if (isPalindrome(number)) {

System.out.println(number + " is a palindrome.");

} else {

System.out.println(number + " is not a palindrome.");

}

}

}

**Explanation:**

1. The program takes an integer as input.
2. It checks for special cases (negative numbers, numbers ending with 0).
3. It reverses the number digit by digit.
4. It compares the reversed number with the original number.

**Output:**

121 is a palindrome.

**Time and Space Complexity:**

* Time complexity: O(log n), where n is the input number.
* Space complexity: O(1), as the program uses constant space for variables.

**10. Leap Year:**

**Program:**

Java

import java.util.Scanner;

public class LeapYear {

public static boolean isLeapYear(int year) {

return (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter

a year: ");

int year = scanner.nextInt();

if (isLeapYear(year)) {

System.out.println(year + " is a leap year.");

} else {

System.out.println(year + " is not a leap year.");

}

}

}

**Explanation:**

1. The program takes a year as input.
2. It checks the leap year conditions:
   * The year is divisible by 4 but not by 100.
   * The year is divisible by 400.
3. If either condition is true, the year is a leap year.

**Output:**

Enter a year: 2020

2020 is a leap year.

**Time and Space Complexity:**

* Time complexity: O(1), as the program performs constant calculations.
* Space complexity: O(1), as the program uses constant space for variables.

Sources and related content