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.

```
package com.assignment;
import java.util.Scanner;

public class Solution1 {
public static boolean isArmstrong(int number) {
int originalNumber = number, remainder, result = 0;
int digits = String.valueOf(number).length(); // Count the number of
digits
while (originalNumber != 0) {
remainder = originalNumber % 10;
result += Math.pow(remainder, digits);
originalNumber /= 10;
}
```

```java
return result == number;
}
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.println("Enter a number:");
int number = sc.nextInt();
System.out.println("Is Armstrong: " + isArmstrong(number));
}
}
```

Explanation:

- Input a number.
- Raise each digit to the power of the total number of digits and sum them.
- If the sum equals the original number, it’s an Armstrong number.

Output:

- **Input**: 153

Output: `true`

- **Input**: 123

Output: `false`

Time Complexity:

- **Time**: O(d), where d is the number of digits.
- **Space**: O(1), constant space.

2. Prime Number

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

```
package com.assignment;


import java.util.Scanner;

public class Solution2 {
public static boolean isPrime(int num) {
if (num <= 1) return false;
for (int i = 2; i <= Math.sqrt(num); i++) {
if (num % i == 0) {
return false;
}
}
return true;
}
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.println("Enter a number:");
int num = sc.nextInt();
System.out.println("Is Prime: " + isPrime(num));
}
}
```

Explanation:

- Input a number.
- Check if it's divisible by any number between 2 and the square root of the number.
- If it is, return false; otherwise, return true.

INPUT-OUTPUT

```
Enter a number:
29
Is Prime: true
Enter a number:
15
```

```
Is Prime: false
```

Flowchart:

1. Start
2. Input: num
3. Check divisibility from 2 to sqrt(num)
4. If divisible return false
5. If not, return true
6. End

Time Complexity:

- **Time**: O(√n), because we only check divisors up to √n.
- **Space**: O(1), constant space.

3. Factorial

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

```java
package com.assignment;

import java.util.Scanner;

public class Solution3 {
public static int factorial(int num) {
if (num == 0 || num == 1) return 1;
int fact = 1;
for (int i = 2; i <= num; i++) {
fact *= i;
}
return fact;
}
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.println("Enter a number:");
int num = sc.nextInt();
System.out.println("Factorial: " + factorial(num));
}
}
```

Output:

```
Enter a number:
5
Factorial: 120
Enter a number:
0
Factorial: 1
```

```
Flowchart:
[Start] -> [Input: num] -> [Initialize fact=1]
     |
[Loop from 2 to num] -> [Multiply fact by i]
     |
[Return fact] -> [End]
```

Explanation:

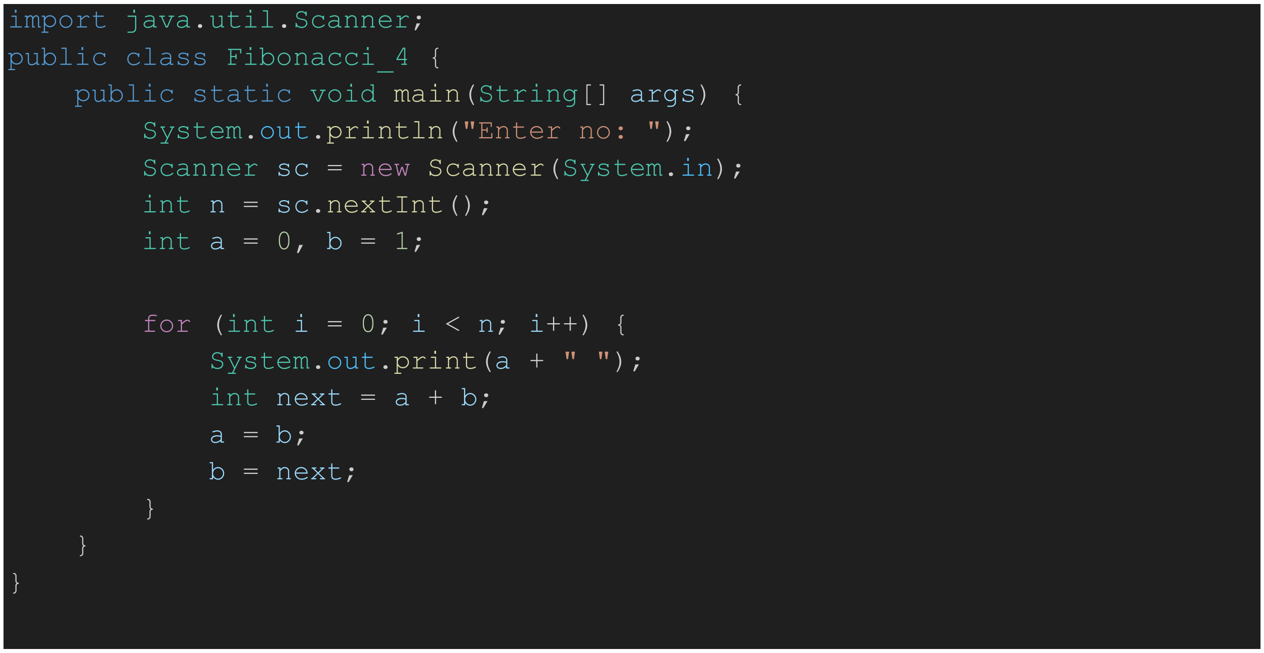
- Input a number.
- Multiply all numbers from 2 to the input number to compute the factorial.

Time Complexity:

- **Time**: O(n), where n is the number.
- **Space**: O(1), constant space.

4. Fibonacci Series

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



```
import java.util.Scanner;
public class Fibonacci_4 {
    public static void main(String[] args) {
        System.out.println("Enter no: ");
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int a = 0, b = 1;

        for (int i = 0; i < n; i++) {
            System.out.print(a + " ");
            int next = a + b;
            a = b;
            b = next;
        }
    }
}
```

Flowchart

1. **Start**:
    a. The program starts.
2. **Ask for Input**:
    a. The program asks the user to enter a number (n).
3. **Get Input**:
    a. The user enters a number, and the program reads it.
4. **Set Initial Values**:
    a. Set the first two numbers of the Fibonacci sequence: a = 0 and b = 1.
5. **Loop Start**:
    a. The program starts a loop that will run n times.
6. **Print the Number**:
    a. The program prints the current value of a.
7. **Calculate the Next Fibonacci Number**:
    a. Calculate the next number in the Fibonacci sequence by adding a + b.
8. **Update Values**:
    a. Set a to the value of b, and set b to the new Fibonacci number (the one you just calculated).

9. **Repeat**:
    a. The loop repeats until the program has printed n Fibonacci numbers.
10. **End**:
    a. Once the loop finishes, the program ends.

Output

```
Enter no:

5

0 1 1 2 3

Enter no:

8

0 1 1 2 3 5 8 13
```

Time Complexity: O(n) **Space Complexity:** O(1)

5. Find GCD

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

```java
package com.assignment;

import java.util.Scanner;

public class Solution5 {
public static int gcd(int a, int b) {
if (b == 0) return a;
return gcd(b, a % b);
}
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
try {
System.out.println("Enter two numbers:");
int a = sc.nextInt();
int b = sc.nextInt();
System.out.println("GCD: " + gcd(a, b));
} finally {
sc.close();
}
}
}
```

Explaination:

- Input two numbers.
- Use the Euclidean algorithm to find the GCD by repeatedly taking the remainder of division.

```
Enter two numbers:
24
54
GCD: 6

Enter two numbers:
17
13
GCD: 1
```

- **Time**: O(log(min(a, b))), where a and b are the two numbers.
- **Space**: O(1), constant space.

Flowchart:
1. Start: input a and b from the user
2. Call gcd(a,b)
3. Check if b==0: if b is 0, then base case is reached & function returns a.
   Yes: If b==0, the GCD is a & the function returns the result.
   No: If b!=0, the function calls itself recursively with the arguments gcd (b, a%b).
4. Recursion: continues until b becomes 0, and the GCD is found.
5. End

6. Find Square Root

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

```
package com.assignment;

import java.util.Scanner;

public class Solution6 {
public static int findSquareRoot(int x) {
if (x == 0 || x == 1) {
return x;
}
int start = 1, end = x, result = 0;
while (start <= end) {
int mid = (start + end) / 2;

if (mid * mid == x) {
return mid;
}

if (mid * mid < x) {
start = mid + 1;
result = mid;
} else {
end = mid - 1;
}
}
return result;
}
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.println("Enter a number to find its square root:");
int x = sc.nextInt();
System.out.println("Square Root: " + findSquareRoot(x));
sc.close();
}
}
```

```
Flowchart:

[Start] -> [Input x] -> [Check if x == 0 or 1]
     |                |
     Yes               No
     |                |
[Return x]      [Binary Search for Square Root] -> [Output Result]
     |                                               |
   [End]                                          [End]
```

Explanation:

- This program finds the square root using **binary search** between 1 and the number x.
- We start with `start = 1` and `end = x`. If `mid * mid == x`, we return `mid`. Otherwise, adjust the search space based on whether `mid * mid` is less than or greater than x.
- This approach gives an integer approximation of the square root.

```
Enter a number to find its square root:
16
Square Root: 4
27
Square Root: 5
```

Time and Space Complexity:

- **Time Complexity**: `O(log x)` — due to the binary search over the range [1, x].
- **Space Complexity**: `O(1)` — no additional space is used except variables.

7. Find Repeated Characters in a String

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

```java
package com.assignment;


import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

public class Solution7 {
public static void findRepeatedCharacters(String str) {
Map<Character, Integer> charCountMap = new HashMap<>();
for (char c : str.toCharArray()) {
charCountMap.put(c, charCountMap.getOrDefault(c, 0) + 1);
}
System.out.println("Repeated characters are:");
for (Map.Entry<Character, Integer> entry : charCountMap.entrySet()) {
if (entry.getValue() > 1) {
System.out.print(entry.getKey() + " ");
}
}
System.out.println();
}

public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.println("Enter a string:");
String str = sc.nextLine();
findRepeatedCharacters(str);
sc.close();
}
}
```

```
OUTPUT:
Enter a string:
programming
Repeated characters are:
r g m

hello
Repeated characters are:
L
```

Explanation:

- The program uses a **HashMap** to count the occurrences of each character in the string.
- It iterates over the string, adding each character to the map and incrementing its count. After that, it prints the characters that appear more than once.

Flowchart:

```
1. Start
2. Input string
3. For each character in string
4. Create HashMap
5. Increment count for each character
6. Print repeated characters
7. End
```

- **Time Complexity**: O(n) — iterating over the string once.
- **Space Complexity**: O(n) — storing character counts in a HashMap

8. First Non-Repeated Character

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

```
package com.assignment;

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

public class Solution8 {
public static Character findFirstNonRepeated(String str) {
// LinkedHashMap preserves the order of insertion
Map<Character, Integer> charCountMap = new LinkedHashMap<>();
// Iterate through the string and populate the map with character
counts
for (char c : str.toCharArray()) {
charCountMap.put(c, charCountMap.getOrDefault(c, 0) + 1);
}
// Iterate through the map to find the first character with a count of
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 sc = new Scanner(System.in);
System.out.println("Enter a string:");
String str = sc.nextLine();
// Find the first non-repeated character in the string
Character result = findFirstNonRepeated(str);
if (result != null) {
System.out.println("First non-repeated character: " + result);
} else {
System.out.println("No non-repeated character found.");
}
sc.close();
}
}
```

OUTPUT:

```
Enter a string:
stress
First non-repeated character: t
aabbcc

No non-repeated character found.
```

Explanation:

- The program uses a **LinkedHashMap** to maintain the insertion order of characters. It first counts occurrences, and then checks for the first character with a count of 1 (non-repeated).

Flowchart:

1. Start
2. Input String from User
3. Initialize LinkedHashMap (charCountMap)
4. For each character in the string
5. Add to map with frequency count
6. Find the first entry with count 1
7. Output the result or null
8. End

Time complexity is O(n)

Space complexity is O(n) where n is the number of characters in the string.

9. Integer Palindrome

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

```
import java.util.Scanner;

class IntPalindrome {
public static void main (String [] args){
Scanner scan = new Scanner(System.in);
System.out.println("Enter The Number To Check If It's a Palindrome or Not..");
int num = scan.nextInt();
int Revnum = 0;
int temp = num;
while (temp !=0) {
int digit = temp % 10;
Revnum = Revnum *10 + digit;
temp = temp/10;
}
if (num == Revnum) {
System.out.println(" Yes, "+ num +" is a palindrome number");
              } else {
     System.out.println(" No, "+ num +" is not a palindrome number");
}
}
}
```

Input: 121

Output: true

Input: -121

Output: false

Time Complexity : $O(\log 10)$
Space Complexity : O(1)

Flowchart:

Start: Begin the algorithm.
Input Number: Prompt the user to enter a number and read the input value into the variable num.
Initialize Variables:
Set Revnum to 0 (this will hold the reversed number).
Set temp to num (this will be used to extract digits).
Reverse the Number:
While temp is not equal to 0, repeat the following steps:
Extract the last digit of temp using digit = temp % 10.
Update Revnum by multiplying it by 10 and adding the extracted digit: Revnum = Revnum * 10 + digit.
Remove the last digit from temp by performing integer division by 10: temp = temp / 10.
Check for Palindrome:
If num is equal to Revnum, then:
Print "Yes, num is a palindrome number."
Otherwise:
Print "No, num is not a palindrome number."
End: Terminate the algorithm.

10. Leap Year

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

```
package com.assignment;


import java.util.Scanner;

public class Solution10 {
public static boolean isLeapYear(int year) {
if (year % 4 == 0) {
```

```java
if (year % 100 == 0) {
if (year % 400 == 0) {
return true;
} else {
return false;
}
} else {
return true;
}
}
return false;
}

public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
System.out.println("Enter a year:");
int year = sc.nextInt();
System.out.println("Is Leap Year: " + isLeapYear(year));
}
}
```

Output:

```
Enter a year:
2020
Is Leap Year: true
Enter a year:
1900
Is Leap Year: false
```

Flowchart:

```
[Start] -> [Input: year] -> [Check if divisible by 4]
     |
[Check if divisible by 100] -> [Check if divisible by 400]
     |
[Return true/false] -> [End]
```

Explanation:

1. Input the year.
2. Check if the year is divisible by 4.
3. If it is divisible by 100, check if it is divisible by 400.
4. Return true if it satisfies the conditions for being a leap year.

Time Complexity:

- **Time**: O(1), constant time.
- **Space**: O(1), constant space.