I have now finished the challenge. I am going to show type of processing involved for the test cases given. Note I have included the decimal and fract7ional component as presented test case, in order for the code to verify it was end user intention....
NOTE: I have not performed exactly to the challenge requirements since it states that precision component is repeating.. I have rather completed this as an exercise to validate data entry..... Since in real life, decimals are interchanged with fractions...

Test case 1

***** The Decimal presented: 0.6**

The fraction part: 6

this is numerator: 6 **//as expected**

this is denominator: 10 **//as expected**

reduced numerator: 3 **//correctly been reduced**

reduced denominator: 5 **//correctly been reduced**

Fraction presented is: 2/3 **//this is as provided by end user**

Fraction provided in challenge: 2/3 might not be correctly reduced (3/5)
//since it has converted the decimal 0.6 to fraction, it clearly will not get 2/3.. It gets 3/5 as expected

2Original reduced fraction provided 2/3 is non- representation of the initial decimal conversion: 0.6
//first point it clarifies to end user that fraction is a mismatch to the decimal

1Remaining original + fraction(2/3) in decimal is 0.6666666666666666 and NOT:0.6
//it now performs the conversion of original fraction to decimal

****

This will just be final part of the code...

Since several prime numbers have been generated, I will use the existing numerator

And store the results in a String array: Numerator/prime number AS fraction and Numerator/prime number AS decimal and store results. IT IS TO GET A FEEL OF DIFFERENT PATTERNS AS EXPLORED IN DOCUMENTATION!

This will provide a sample should I wish to populate initial array also!
//The message above is just an additional exercise to generate sample of further numbers which can be filled in TestOne array

****

CHECK TO SEE IF ORIGINAL FRACTION IS TERMINATING:

NOT Terminating fraction since reduced denominator is NOT divisible by prime factor of two or five

Original numerator / PRIME number AS FRACTION (not reduced) => **2/2**
//dividing by prime number of 2 creates a terminating fraction as expected. We know performing on numerator will have same effect on the denominator since both have been reduced as part of the exercise.

Original numerator / PRIME number AS DECIMAL => 1.0
//this is conversion of the above

Original numerator / PRIME number AS FRACTION (not reduced) => 2/3
//dividing by prime number of 3 creates a non-terminating (recurring) fraction as expected
Original numerator / PRIME number AS DECIMAL => 0.6666666666666666
//this is conversion of the above. Note it is limited to precision available in double. I can potentially try to venture into BigDecimal... But I found it excessive, since if this number was to be used in TestOne array, it would need to be reduced in size. I found in the code, Java could not handle the reduction if the decimal was 0.800000000
But it could handle: 0.750000000
I had option either to reduce numerator to below accepted threshold... But I felt that would be detrimental to calculation.. INSTEAD, I opted to reduce the precision (it will inform end user).... It will become actually clearer with a test case of a lengthy decimal number...

Original numerator / PRIME number AS FRACTION (not reduced) => 2/5
//Again just like prime number 2, it renders a terminating fraction..

Original numerator / PRIME number AS DECIMAL => 0.4

Original numerator / PRIME number AS FRACTION (not reduced) => 2/7

Original numerator / PRIME number AS DECIMAL => 0.2857142857142857

Original numerator / PRIME number AS FRACTION (not reduced) => 2/11

Original numerator / PRIME number AS DECIMAL => 0.18181818181818182

Original numerator / PRIME number AS FRACTION (not reduced) => 2/13

Original numerator / PRIME number AS DECIMAL => 0.15384615384615385

Original numerator / PRIME number AS FRACTION (not reduced) => 2/17

Original numerator / PRIME number AS DECIMAL => 0.11764705882352941

Original numerator / PRIME number AS FRACTION (not reduced) => 2/19

Original numerator / PRIME number AS DECIMAL => 0.10526315789473684

Original numerator / PRIME number AS FRACTION (not reduced) => 2/23

Original numerator / PRIME number AS DECIMAL => 0.08695652173913043

Original numerator / PRIME number AS FRACTION (not reduced) => 2/29

Original numerator / PRIME number AS DECIMAL => 0.06896551724137931

Original numerator / PRIME number AS FRACTION (not reduced) => 2/31

Original numerator / PRIME number AS DECIMAL => 0.06451612903225806

Original numerator / PRIME number AS FRACTION (not reduced) => 2/37
I have cut the outputs, but I have let it continue up to variable long a = 2000.. This is a complete user choice...

Test case 2

*** The Decimal presented: 1.1

The fraction part: 1

this is numerator: 1

this is denominator: 10

reduced numerator: 1 **//it has calculated 0.1 as 1/10 as expected. It can be seen it has not been able to reduce numerator and denominator above (1/10)**

reduced denominator: 10

Fraction presented is: 10/9 **//this is correct fraction**

Original numerator/denominator:1

remaining numerator: 1

remiaining fraction: 1/9

Fraction provided in challenge is improper: 10/9 //it has identified this from larger numerator than denominator

Improper fraction with no existing whole number in front //acknowledges no whole number in front

Proper entire fraction is: 1-1/9 //the portion before the - is the whole number

Fraction provided in challenge: 1/9 might not be correctly reduced (1/10) //it is identifying this since we know 1/9 is 0.111111111111 (recurring)...

1Original fraction(10/9)has been changed to proper fraction(1-1/9). Remaining fraction is: 1/9 is non- representation of the initial decimal conversion: 0.1 // this is to inform end user... For instance they might have accidentally placed denominator on top of the numerator....

1Remaining fraction 1/9 in decimal is 0.1111111111111111 and NOT:0.1
//again informing end user

****

This will just be final part of the code...

Since several prime numbers have been generated, I will use the existing numerator

And store the results in a String array: Numerator/prime number AS fraction and Numerator/prime number AS decimal and store results. IT IS TO GET A FEEL OF DIFFERENT PATTERNS AS EXPLORED IN DOCUMENTATION!

This will provide a sample should I wish to populate initial array also!

****

CHECK TO SEE IF ORIGINAL FRACTION IS TERMINATING:

Terminating fraction since reduced denominator is divisble by prime number of two or five
//from above we can see reduced denominator is 10. And we know it is divisible by prime number 2 or 5.

1.1 is a periodic number
//if it is terminating, it informs end user that 1.1 is periodic number

//AGAIN, this just divides the original numerator with prime numbers...
//We can see how the repetent varies in length...

Original numerator / PRIME number AS FRACTION (not reduced) => 10/2

Original numerator / PRIME number AS DECIMAL => 5.0

Original numerator / PRIME number AS FRACTION (not reduced) => 10/3

Original numerator / PRIME number AS DECIMAL => 3.3333333333333335

Original numerator / PRIME number AS FRACTION (not reduced) => 10/5

Original numerator / PRIME number AS DECIMAL => 2.0

Original numerator / PRIME number AS FRACTION (not reduced) => 10/7

Original numerator / PRIME number AS DECIMAL => 1.4285714285714286

Original numerator / PRIME number AS FRACTION (not reduced) => 10/11

Original numerator / PRIME number AS DECIMAL => 0.9090909090909091

Original numerator / PRIME number AS FRACTION (not reduced) => 10/13

Original numerator / PRIME number AS DECIMAL => 0.7692307692307693

Original numerator / PRIME number AS FRACTION (not reduced) => 10/17

Original numerator / PRIME number AS DECIMAL => 0.5882352941176471

Original numerator / PRIME number AS FRACTION (not reduced) => 10/19

Test case 2a

Exploring again, but this time trying a mismatch between the whole number in decimal and while number once converted to a proper fraction....

*** The Decimal presented: 6.1

The fraction part: 1

this is numerator: 1

this is denominator: 10

reduced numerator: 1

reduced denominator: 10

Fraction presented is: 10/9

Original numerator/denominator:1

remaining numerator: 1

remiaining fraction: 1/9

Fraction provided in challenge is improper: 10/9

Improper fraction with no existing whole number in front

Proper entire fraction is: 1-1/9

*****Mismatch in whole numbers****** **//it shows the mismatch here**

Fraction has following whole number:1

Original decimal has following whole number:6
****

Fraction provided in challenge: 1/9 might not be correctly reduced (1/10)

1Original fraction(10/9)has been changed to proper fraction(1-1/9)

Remaining fraction is: 1/9 is non- representation of the initial decimal conversion: 0.1

1Remaining fraction 1/9 in decimal is 0.1111111111111111 and NOT:0.1

Test case 2b

Exactly same as 2a… But this time experimenting with a mismatch in whole number between a decimal and a proper fraction.

```
testOne[0]="6.1,
testOne[1]="1-3/9,
```

It has not shown the mismatch code since it was exclusively in area of improper fraction. I will try to copy this piece of code and adapt it:

Currently in improper section

```
697
698   if (String.valueOf(denominatorIntoNumerator+wholeNumberBeforeFractionLong)!=wholeNumberPortionDecimal)
699   {
700       System.out.println("***Mismatch in whole numbers****");
701       System.out.println("Fraction has following whole number:" + (denominatorIntoNumerator+wholeNumberBeforeFracti
702       System.out.println("Original decimal has following whole number:" + wholeNumberPortionDecimal);
703       System.out.println("**********");
704   }
```

Modified to:

```
730       if (!improperFraction)
731       {
732          if (String.valueOf(wholeNumberBeforeFractionLong)!=wholeNumberPortionDecimal)
733         {
734            System.out.println("***2Mismatch in whole numbers****");
735            System.out.println("Fraction has following whole number:" + (wholeNumberBeforeFractionLong));
736            System.out.println("Original decimal has following whole number:" + wholeNumberPortionDecimal);
737            System.out.println("**********");
738       }
739       }
```

*** The Decimal presented: 6.1

The fraction part: 1

this is numerator: 1

this is denominator: 10

reduced numerator: 1

reduced denominator: 10

Fraction presented is: 1-3/9

2Mismatch in whole numbers*

Fraction has following whole number:1

Original decimal has following whole number: 6

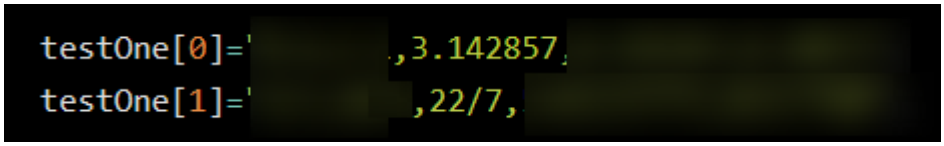
****

2Fraction provided in challenge: 1-3/9 might not be correctly reduced (1/10)

2Original reduced fraction provided 1-3/9 is non- representation of the initial decimal conversion: 0.1

1Remaining original fraction(1-3/9) in decimal is 0.3333333333333333 and NOT:0.1

Test case 3



***** The Decimal presented: 3.142857**

The fraction part: 142857

this is numerator: 142857

this is denominator: 1000000

reduced numerator: 142857 //it can be seen reduction is not possible

reduced denominator: 1000000

Fraction presented is: 22/7

Original numerator/denominator:3 //this is the process for it creating a proper fraction from improper fraction

remaining numerator: 1

remiaining fraction: 1/7

Fraction provided in challenge is improper: 22/7

Improper fraction with no existing whole number in front //acknowledged no whole number in front

Proper entire fraction is: 3-1/7 //correctly calculated this

Fraction provided in challenge: 1/7 might not be correctly reduced (142857/1000000)

1Original fraction(22/7)has been changed to proper fraction(3-1/7). Remaining fraction is: 1/7 is non- representation of the initial decimal conversion: 0.142857

1Remaining fraction 1/7 in decimal is 0.142857142857142857 and NOT:0.142857
//This is the first example we can see visually (as per challenge) that repetend is spanning the entire decimal range (when pi is approximated to 22/7 or 3 - 1/7).
In Ideal world, this would be the best chance to look out for pattern..... But on other hand, we can see the precision is 17 digits wide... We know from my research that repetend can reach thousands of digits wide... So it would be possible to pick this up from the screen output..
//Perhaps in the future, I can perform regex / pattern search for the decimal component in the final output 0.142857142857142857. I can then inform end user that the recurring has occurred on the following digits: 142857

```
fractions("3.(142857)") → "22/7"
```

****

This will just be final part of the code...

Since several prime numbers have been generated, I will use the existing numerator

And store the results in a String array: Numerator/prime number AS fraction and Numerator/prime number AS decimal and store results. IT IS TO GET A FEEL OF DIFFERENT PATTERNS AS EXPLORED IN DOCUMENTATION!

This will provide a sample should I wish to populate initial array also!

****

CHECK TO SEE IF ORIGINAL FRACTION IS TERMINATING:

Terminating fraction since reduced denominator is divisble by prime number of two or five
//we know reduced denominator is 1000000.....

3.142857 is a periodic number
//again we know that it is henceforth a periodic number...
//but we have also seen that if the original fraction was utilized, the fractional part depicts irrational number... And we know that pi is irrational and hence using 22/7 is the approximation.

Original numerator / PRIME number AS FRACTION (not reduced) => 22/2

Original numerator / PRIME number AS DECIMAL => 11.0

Original numerator / PRIME number AS FRACTION (not reduced) => 22/3

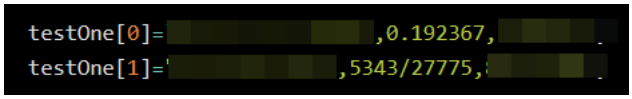
Original numerator / PRIME number AS DECIMAL => 7.333333333333333

Original numerator / PRIME number AS FRACTION (not reduced) => 22/5

Original numerator / PRIME number AS DECIMAL => 4.4

Original numerator / PRIME number AS FRACTION (not reduced) => 22/7

Test case 4



*** The Decimal presented: 0.192367

The fraction part: 192367

this is numerator: 192367 //again can be seen, no reduction

this is denominator: 1000000

reduced numerator: 192367

reduced denominator: 1000000

Fraction presented is: 5343/27775

Fraction provided in challenge: 5343/27775 might not be correctly reduced (192367/1000000)
//shows discrepancy in the fraction provided by end user and that calculated from the decimal

2Original reduced fraction provided 5343/27775 is non- representation of the initial decimal conversion: 0.192367

1Remaining original + fraction(5343/27775) in decimal is 0.19236723672367237 and NOT:0.192367
//we can see here that there are multiple repetends as expected.

```
fractions("0.19(2367)")
```

Test case 5

*** The Decimal presented: 0.10973

The fraction part: 10973

this is numerator: 10973

this is denominator: 100000

reduced numerator: 10973

reduced denominator: 100000

Fraction presented is: 823/7500

Fraction provided in challenge: 823/7500 might not be correctly reduced (10973/100000)

2Original reduced fraction provided 823/7500 is non- representation of the initial decimal conversion: 0.10973

1Remaining original + fraction(823/7500) in decimal is 0.10973333333333334 and NOT:0.10973
//it can be seen the recurring 3 has been fulfilled

```
fractions("0.1097(3)") → "823/7500"
```

Test case 5: This is a new devised case.. I have developed this so that it does not perform a reduction when there are decimals such as 0.1111111 (if precision length is:

```
int limitPrecision=19;
```

It will use a technique learnt from YouTube on how to readily perform an action...
Note it will be representative of the actual decimal

********Welcome to Online IDE!! Happy Coding :)********** // have now included the following information so end user is aware of the criteria of the decimal and how code handles situation...

NOTE: Precision is limited to 19 digit wide...

NOTE: Since computational issues, reduction on decimal component EXCEEDING 0. 750,000,000 will lose its precision by 1 digit.

EXCEPTION is single digit recurring number

****

*** The Decimal presented: 0.1111111111111111111
Recurring digit occurs: 19 times; //it informs end user of the limit, hence it is considered to be reoccuring through entire precision

Recurring digit through entire decimal: 1

Fraction of recurring: 1/9

Fraction presented is: 1/9

fraction provided in challenge is fully reduced: 1/9 //it identifies that initial numerator and denominator align with initial fraction provided by end user.

Original fraction(1/9) provided is exact representation of the initial decimal conversion: 0.1111111111111111111
//this is based on decimal hitting 19 digit precision and will safely assume single digit 1s

Test case 6: This is a new devised case.. I have developed this so that it does not perform a reduction when there are decimals such as 0.1111111 if the precision length is

```
int limitPrecision=19;
```

It will use a technique learnt from YouTube on how to readily perform an action...
Note: This time, the fraction will not be in most reduced form (3/27)
FAIL: It just doesn't give the right level of information to end user (see bold areas)...

********Welcome to Online IDE!! Happy Coding :)**********

NOTE: Precision is limited to 19 digit wide...

NOTE: Since computational issues, reduction on decimal component EXCEEDING 0. 750,000,000 will lose its precision by 1 digit.

EXCEPTION is single digit recurring number

****

*** The Decimal presented: 0.1111111111111111111

Recurring digit occurs: 19 times //it informs end user of meeting limitPrecision of 19

Recurring digit through entire decimal: 1

Fraction of recurring: 1/9

Fraction presented is: 3/27

Fraction provided in challenge: 3/27 might not be correctly reduced (1/9) //informs end user fraction not reduced

2Original reduced fraction provided 3/27 is non- representation of the initial decimal conversion: 0.1111111111111111111 //does not seem applicable to recurring scenario, so I have added logic to display for recurring single digit scenario also

1Remaining original + fraction(3/27) in decimal is 0.1111111111111111 and NOT:0.1111111111111111111 //does not seem applicable to recurring scenario, so I have added logic to display for recurring single digit scenario also. Not the 0.11111111111111111111 (19 digits precision), is directly from the String, hence has more than 0.1111111111111111 (16 digits).

Test 6a

I visited all of my previous test cases, notably from Test 2a onwards... And increased the intensity of my screen outputs to cater for improper and proper fractions.... And mismatch in the whole numbers...
It was then I could attempt Test 6 again...

*** The Decimal presented: 0.1111111111111111111

Recurring digit occurs: 19 times

Recurring digit through entire decimal: 1

Fraction of recurring: 1/9

Fraction presented is: 3/27

Improper fraction with no existing whole number in front

2Fraction provided in challenge: 3/27 might not be correctly reduced (1/9)

2Original reduced fraction provided 3/27 is non- representation of the initial decimal conversion: 0.1111111111111111111

Test case 7: This is a new devised case.. I have developed this so that it does not perform a reduction when there are decimals such as 7.1111111 if the precision length is `int limitPrecision=19;`

It will use a technique learnt from YouTube on how to readily perform an action...

Note: This time, the fraction will be 7 - 1/9

********Welcome to Online IDE!! Happy Coding :)**********

NOTE: Precision is limited to 19 digit wide...

NOTE: Since computational issues, reduction on decimal component EXCEEDING 0. 750,000,000 will lose its precision by 1 digit.

EXCEPTION is single digit recurring number

****

*** The Decimal presented: 7.1111111111111111111

This is full equation one: 0.1111111111111111111

Recurring digit occurs: 19 times

Recurring digit through entire decimal: 1

FULL EQUATION ONE: 0.1111111111111111111

FULL EQUATION TWO: 1.1111111111111111111

0.1111111111111111

FULL EQUATION TWO A: 1.0

Fraction of recurring: 1/9

Fraction presented is: 7-1/9

1THE VALUEUUUUUUU: 7-1/9

Var: 7

fraction provided in challenge is fully reduced: 7-1/9

Original fraction(7-1/9) provided is exact representation of the initial decimal conversion: 7.1111111111111111111 **//it can be seen that it is exact conversion..**

Test case 7a: This is a new devised case.. I have developed this so that it does not perform a reduction when there are decimals such as 7.1111111 if the precision length is `int limitPrecision=19;` (FAIL)

It will use a technique learnt from YouTube on how to readily perform an action...

Note: This time, the fraction will also be improper (64/9), but it will render remaining proper fraction as equivalent to 7 - 1/9

*** The Decimal presented: 7.1111111111111111111

This is full equation one: 0.1111111111111111111

Recurring digit occurs: 19 times

Recurring digit through entire decimal: 1

FULL EQUATION ONE: 0.1111111111111111111

FULL EQUATION TWO: 1.1111111111111111111

0.1111111111111111

FULL EQUATION TWO A: 1.0

Fraction of recurring: 1/9

Fraction presented is: 64/9

Improper fraction with no existing whole number in front

Fraction provided in challenge is improper: 64/9 **//it can see it has transitioned improper => proper fraction**

Proper entire fraction is: 7-1/9

*****Mismatch in whole numbers******
//there is a clear error here, the numbers are same but it showing mismatch

```
if (String.valueOf(denominatorIntoNumerator+wholeNumberBeforeFractionLong)!=wholeNumberPortionDecimal)
{
```

I need to be careful that when I compare Strings, I use the correct notation... Instead of above, I changed to the following and issue has passed:

```
if (!String.valueOf(denominatorIntoNumerator+wholeNumberBeforeFractionLong).equals(wholeNumberPortionDecimal))
{
```

I am now visiting my entire code to see if I am comparing Strings and will make similar adjustments...
I will search my code for == or != and examine scenarios closely.

Fraction has following whole number:7

Original decimal has following whole number:7

****

2Fraction provided in challenge: 1/9 might not be correctly reduced (1/9) **//this is not true**

1Original fraction(64/9)has been changed to proper fraction(7-1/9) **//this is correct**

Remaining fraction is: 1/9 is non- representation of the initial decimal conversion: 0.1111111111111111111
//this line has to be dropped for this scenario..

1Remaining fraction 1/9 in decimal is 0.1111111111111111 and NOT:0.1111111111111111111
//this line has to be dropped for this scenario..

As for this error, I did not encounter issue if there was no improper fraction...

Test case 7a 64/9 7.1111111111111111111

It can be seen that this else is associated with if below.....
So I will obtain the key variables:
It can be seen that whilst it has processed the improper fraction route, it has assigned 1/9 to numeratorOriginal.....

```
numeratorOriginal:1/9
numerator:1
denominatorOriginal: 9
denominator:9
```

```
772
773     else
774     {
775         System.out.println("2Fraction provided in challenge: " + temp + " might not be correctly reduced " + "("+numer
776
```

```
2Fraction provided in challenge: 1/9 might not be correctly reduced (1/9)
1Original fraction(64/9)has been changed to proper fraction(7-1/9)
Remaining fraction is: 1/9 is non- representation of the initial decimal conversion: 0.111111111111111111
1Remaining fraction 1/9 in decimal is 0.1111111111111111 and NOT:0.1111111111111111111
```

So the fix is:

Since we know there is single digit recurring in precision, I have added this into the following condition

```
792
793             if (!singleDigitrecurring)
794             {
795
796             System.out.println("Remaining fraction is: " + remainingFraction + " is non- representation of the i
797             System.out.println("1Remaining fraction " + remainingFraction + " in decimal is " + result + " and N
798
799             }
```

Test case 7 7 - 1/9 , 7.1111111111111111111

```
numeratorOriginal::1
numerator:1
denominatorOriginal: 9
denominator:9
```

```
766
767 if (numeratorOriginal.equals(String.valueOf(numerator)) && denominatorOriginal.equals(String.valueOf(denominator)))
768 {
769     System.out.println("fraction provided in challenge is fully reduced: " + temp);
770     System.out.println("Original fraction" + "("+temp+")" + " provided is exact representation of the initial decima
771 }
```

Part of the output

fraction provided in challenge is fully reduced: 7-1/9

Original fraction(7-1/9) provided is exact representation of the initial decimal conversion:
7.1111111111111111111 **//it can be seen that it is exact conversion..**

Test case 7av2: This is a new devised case.. I have developed this so that it does not perform a reduction when there are decimals such as 7.1111111 if the precision length

```
int limitPrecision=19;
```

It will use a technique learnt from YouTube on how to readily perform an action...
Note: This time, the fraction will also be improper (64/9), but it will render remaining proper fraction as equivalent to 7 - 1/9

*** The Decimal presented: 7.111111111111111111

This is full equation one: 0.111111111111111111

Recurring digit occurs: 19 times

Recurring digit through entire decimal: 1

2Fraction provided in challenge: 1/9 might not be correctly reduced (1/9)

Discrepency above is due to conversion from improper=>proper fraction

1Original fraction(64/9)has been changed to proper fraction(7-1/9)

Test case 7b: Same as above, but will use initial decimal as 5.111111111 and see if it identifies mismatch with 64/9 (7 - 1/9)

*** The Decimal presented: 5.111111111111111111

Recurring digit occurs: 19 times

Recurring digit through entire decimal: 1

Improper fraction with no existing whole number in front

Fraction provided in challenge is improper: 64/9

Proper entire fraction is: 7-1/9

*****Mismatch in whole numbers******

Fraction has following whole number:7 **//clearly shown the difference**

Original decimal has following whole number:5 **//clearly shown the difference**

****

2Fraction provided in challenge: 1/9 might not be correctly reduced (1/9)

Discrepency above is due to conversion from improper=>proper fraction

1Original fraction(64/9)has been changed to proper fraction(7-1/9)

****

Test case 7c: Same as above, but will use initial decimal as 5.111111111 and see if it identifies mismatch with 4-64/9 (11 - 1/9)
It has extra challenge of adding the whole number from the fraction to existing.
And it is a recurring single digit scenario also

*** The Decimal presented: 5.1111111111111111111

This is full equation one: 0.1111111111111111111

Recurring digit occurs: 19 times

Recurring digit through entire decimal: 1

Fraction of recurring: 1/9

Fraction provided in challenge is improper: 4-64/9

Proper entire fraction is: 11-1/9

*****Mismatch in whole numbers******

Fraction has following whole number:11 //(it has performed 7+4 as expected)

Original decimal has following whole number:5 //as expected from the decimal 5.11111……

****

2Fraction provided in challenge: 1/9 might not be correctly reduced (1/9)

Discrepency above is due to conversion from improper=>proper fraction

1Original fraction(4-64/9)has been changed to proper fraction(11-1/9)

Test case 8: This is a new devised case.. It will perform a reduce if the fraction is as such 0.33533 since it can identify single digit recurrence on both sides until it has reached a 5. (FAIL)

*** The Decimal presented: 0.33533

The fraction part: 533

this is numerator: 33533

this is denominator: 1000 //issue here

Improper fraction with no existing whole number in front

2Fraction provided in challenge: 33533/100000 might not be correctly reduced (33533/1000) //issue here

2Original reduced fraction provided 33533/100000 is non- representation of the initial decimal conversion: 0.33533

1Remaining original fraction(33533/100000) in decimal is 0.33533 and NOT:0.33533 //issue here

This caused extreme panic, but I realised eventually there was something in the

checkRecurrence() which caused the fractionalPart to shorten…

```
//0.7777777777777787777
System.out.println("1FRACTIONAL!!!: " + fractionalPart);
checkRecurrence();
System.out.println("2FRACTIONAL!!!: " + fractionalPart);
```

```
1FRACTIONAL!!!: 33533
2FRACTIONAL!!!: 533
```

I expect this to be an isolated incident since it did not affect decimals such as 0.333333333
So once again, I am extremely glad I have tried this scenario.

It can be seen this is correct if it anticipated all digits to be recurring. The fractionalPart is discarding its first digit each time it finds same digit at the end.
We know this is not the case, so it will need to enter else statement when it identifies 33**5**33

```
if (fractionalPart.charAt(0)==fractionalPart.charAt(fractionalPart.length()-1))
{
    //System.out.println("IN HERE11");
    numberConsecutiveDigits++;

    singleDigitrecurring = true;

    fractionalPart=fractionalPart.substring(1);

    partialEquationTwo = partialEquationTwo + fractionalPart.charAt(0);


}
```

But looking at the else, I had correct mentality, but I did not re-instate the fractionalPart

```
else
{
    //it needs this state to set it back to false..
    //since if there is a fraction such as 0.343
    //it would initially enter if due to digit 3 at front and end.
    //so singleDigitrecurring=true...  But with the 4 in middle, it prevents it being a single
    //System.out.println("NOT HERE2222");
    singleDigitrecurring = false;
    break;
}
```

```
public void checkRecurrence()
{
   String backupFractionalPart = fractionalPart;

    if(fractionalPart.length()>1)
    {
```

```
else
{
    //it needs this state to set it back to false..
    //since if there is a fraction such as 0.343
    //it would initially enter if due to digit 3 at front and e
    //so singleDigitrecurring=true...  But with the 4 in middle
    //System.out.println("NOT HERE2222");
    fractionalPart=backupFractionalPart;
    singleDigitrecurring = false;
    break;
```

*** The Decimal presented: 0.33533

fractional part before method call: 33533

fractional part after method call: 33533

The fraction part: 33533

this is numerator: 33533

this is denominator: 100000

reduced numerator: 33533

reduced denominator: 100000

Fraction presented is: 33533/100000

Improper fraction with no existing whole number in front

fraction provided in challenge is fully reduced: 33533/100000

Original fraction(33533/100000) provided is exact representation of the initial decimal conversion: 0.33533

Test case 9: New devised case. This will ensure if the decimal is too big **(greater than 0.750 000 000)**, the code will automatically truncate the LSD (least significant digit)
In practice, this will affect the entire calculation... Strictly speaking we could also take off similar number of digits from the denominator (on the basis that is a proper fraction). It is difficult to factor this into the decimal in test case. But this is the first type of example where there will be a noticeable difference in decimal output. This might be a good technique for analysis purpose...

*** The Decimal presented: 0.98700000259999948

fractional part before method call: 98700000259999948

fractional part after method call: 98700000259999948

This is full equation one: 0.98700000259999948

this is numerator: 98700000259999948

this is denominator: 100000000000000000

Denominator(100000000000000000) considered too large for computation

Denominator reduced to 9 digits wide: 10000000 //denominator has been reduced to support reduction.... //the denominator is only reduced if it is larger or same as numerator.. Otherwise, it would significantly affect the fractional component if for instance it was improper fraction...

Numerator(98700000259999948) considered too large for computation

Numerator reduced to 9 digits wide: 98700000 **//numerator has also been reduced to 9 digits. To preserve integrity of final calculation...**

reduced numerator: 987

reduced denominator: 100

Fraction presented is: 987000200/1000000200

Improper fraction with no existing whole number in front //this is correct

2Fraction provided in challenge: 987000200/1000000200 might not be correctly reduced (987/100) //we know it has not been correctly reduced... And reason is since both numerator and denominator have been truncated

2Original reduced fraction provided 987000200/1000000200 is non- representation of the initial decimal conversion: 0.98700000259999948 //we know this is the case since we had to unfortunately truncate the decimal before putting it into the test variable... Otherwise Java would throw an error

1Remaining original fraction(987000200/1000000200) in decimal is 0.9870000025999994 and NOT:0.98700000259999948

//we can see the difference of truncating digits
987000200/1000000200 = 0.9870000025999994 **(marginally less precision due to limitation of 16 digit precision)**
The original decimal is: 0.98700000259999948 **(this has freedom of approx. 18 or digit precision in test cases).**