**6 kyu**

**Irreducible Sum of Rationals**

55687% of 16441 of878[g964](https://www.codewars.com/users/g964)

C#

* [TRAIN AGAIN](https://www.codewars.com/kata/irreducible-sum-of-rationals/train/csharp)
* [NEXT KATA](https://www.codewars.com/trainer/csharp)

Details

[Solutions](https://www.codewars.com/kata/irreducible-sum-of-rationals/solutions/csharp)

[Forks (3)](https://www.codewars.com/kata/irreducible-sum-of-rationals/forks/csharp)

[Discourse (77)](https://www.codewars.com/kata/irreducible-sum-of-rationals/discuss/csharp)

* Add to Collection
* |
* Share this kata:

You will have a list of rationals in the form

lst = [ [numer\_1, denom\_1] , ... , [numer\_n, denom\_n] ]

or

lst = [ (numer\_1, denom\_1) , ... , (numer\_n, denom\_n) ]

where all numbers are positive integers. You have to produce their sum N / D in an irreducible form: this means that N and D have only 1 as a common divisor.

Return the result in the form:

* [N, D] in Ruby, Python, Clojure, JS, CS, PHP, Julia
* Just "N D" in Haskell
* "[N, D]" in Java, CSharp, TS, Scala, PowerShell
* {N, D} in C++, Elixir
* Some((N, D)) in Rust
* Some "N D" in F#, Ocaml
* c(N, D) in R

If the result is an integer (D evenly divides N) return:

* an integer in Ruby, Elixir, Clojure, Python, JS, CS, PHP, R, Julia
* Just "n" (Haskell)
* "n" Java, CSharp, TS, Scala, PowerShell
* {n, 1} in C++, C
* Some((n, 1)) in Rust
* Some "n" in F#, Ocaml,
* (n, 1) in Swift

If the input list is empty, return nil/None/null/Nothing (or {0, 1} in C++, C) (or "0" in Scala, PowerShell)

**Example:**

[ [1, 2], [1, 3], [1, 4] ] --> [13, 12]

1/2 + 1/3 + 1/4 = 13/12

<https://www.codewars.com/kata/irreducible-sum-of-rationals/csharp>

/\*

\* Author: Syed Mehroz Alam

\* Email: smehrozalam@yahoo.com

\* URL: Programming Home "http://www.geocities.com/smehrozalam/"

\* Date: 6/15/2004

\* Time: 10:54 AM

\*

\*/

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// Classes Contained:

/// Fraction

/// FractionException

/// </summary>

/// Class name: Fraction

/// Developed by: Syed Mehroz Alam

/// Email: smehrozalam@yahoo.com

/// URL: Programming Home "http://www.geocities.com/smehrozalam/"

/// Version: 2.0

///

/// What's new in version 2.0:

/// \* Changed Numerator and Denominator from Int32(integer) to Int64(long) for increased range

/// \* renamed ConvertToString() to (overloaded) ToString()

/// \* added the capability of detecting/raising overflow exceptions

/// \* Fixed the bug that very small numbers e.g. 0.00000001 could not be converted to fraction

/// \* Other minor bugs fixed

///

/// What's new in version 2.1

/// \* overloaded user-defined conversions to/from Fractions

///

///

/// Properties:

/// Numerator: Set/Get value for Numerator

/// Denominator: Set/Get value for Numerator

/// Value: Set an integer value for the fraction

///

/// Constructors:

/// no arguments: initializes fraction as 0/1

/// (Numerator, Denominator): initializes fraction with the given numerator and denominator values

/// (integer): initializes fraction with the given integer value

/// (long): initializes fraction with the given long value

/// (double): initializes fraction with the given double value

/// (string): initializes fraction with the given string value

/// the string can be an in the form of and integer, double or fraction.

/// e.g it can be like "123" or "123.321" or "123/456"

///

/// Public Methods (Description is given with respective methods' definitions)

/// (override) string ToString(Fraction)

/// Fraction ToFraction(string)

/// Fraction ToFraction(double)

/// double ToDouble(Fraction)

/// Fraction Duplicate()

/// Fraction Inverse(integer)

/// Fraction Inverse(Fraction)

/// ReduceFraction(Fraction)

/// Equals(object)

/// GetHashCode()

///

/// Private Methods (Description is given with respective methods' definitions)

/// Initialize(Numerator, Denominator)

/// Fraction Negate(Fraction)

/// Fraction Add(Fraction1, Fraction2)

///

/// Overloaded Operators (overloaded for Fractions, Integers and Doubles)

/// Unary: -

/// Binary: +,-,\*,/

/// Relational and Logical Operators: ==,!=,<,>,<=,>=

///

/// Overloaded user-defined conversions

/// Implicit: From double/long/string to Fraction

/// Explicit: From Fraction to double/string

/// </summary>

class Program

{

public class Fraction

{

/// <summary>

/// Class attributes/members

/// </summary>

long m\_iNumerator;

long m\_iDenominator;

/// <summary>

/// Constructors

/// </summary>

public Fraction()

{

Initialize(0, 1);

}

public Fraction(long iWholeNumber)

{

Initialize(iWholeNumber, 1);

}

public Fraction(double dDecimalValue)

{

Fraction temp = ToFraction(dDecimalValue);

Initialize(temp.Numerator, temp.Denominator);

}

public Fraction(string strValue)

{

Fraction temp = ToFraction(strValue);

Initialize(temp.Numerator, temp.Denominator);

}

public Fraction(long iNumerator, long iDenominator)

{

Initialize(iNumerator, iDenominator);

}

/// <summary>

/// Internal function for constructors

/// </summary>

private void Initialize(long iNumerator, long iDenominator)

{

Numerator = iNumerator;

Denominator = iDenominator;

ReduceFraction(this);

}

/// <summary>

/// Properites

/// </summary>

public long Denominator

{

get

{ return m\_iDenominator; }

set

{

if (value != 0)

m\_iDenominator = value;

else

throw new FractionException("Denominator cannot be assigned a ZERO Value");

}

}

public long Numerator

{

get

{ return m\_iNumerator; }

set

{ m\_iNumerator = value; }

}

public long Value

{

set

{

m\_iNumerator = value;

m\_iDenominator = 1;

}

}

/// <summary>

/// The function returns the current Fraction object as double

/// </summary>

public double ToDouble()

{

return ((double)this.Numerator / this.Denominator);

}

/// <summary>

/// The function returns the current Fraction object as a string

/// </summary>

public override string ToString()

{

string str;

if (this.Denominator == 1)

str = this.Numerator.ToString();

else

str = "["+ this.Numerator + ", " + this.Denominator + "]";

return str;

}

/// <summary>

/// The function takes an string as an argument and returns its corresponding reduced fraction

/// the string can be an in the form of and integer, double or fraction.

/// e.g it can be like "123" or "123.321" or "123/456"

/// </summary>

public static Fraction ToFraction(string strValue)

{

int i;

for (i = 0; i < strValue.Length; i++)

if (strValue[i] == '/')

break;

if (i == strValue.Length) // if string is not in the form of a fraction

// then it is double or integer

return (Convert.ToDouble(strValue));

//return ( ToFraction( Convert.ToDouble(strValue) ) );

// else string is in the form of Numerator/Denominator

long iNumerator = Convert.ToInt64(strValue.Substring(0, i));

long iDenominator = Convert.ToInt64(strValue.Substring(i + 1));

return new Fraction(iNumerator, iDenominator);

}

/// <summary>

/// The function takes a floating point number as an argument

/// and returns its corresponding reduced fraction

/// </summary>

public static Fraction ToFraction(double dValue)

{

try

{

checked

{

Fraction frac;

if (dValue % 1 == 0) // if whole number

{

frac = new Fraction((long)dValue);

}

else

{

double dTemp = dValue;

long iMultiple = 1;

string strTemp = dValue.ToString();

while (strTemp.IndexOf("E") > 0) // if in the form like 12E-9

{

dTemp \*= 10;

iMultiple \*= 10;

strTemp = dTemp.ToString();

}

int i = 0;

while (strTemp[i] != '.')

i++;

int iDigitsAfterDecimal = strTemp.Length - i - 1;

while (iDigitsAfterDecimal > 0)

{

dTemp \*= 10;

iMultiple \*= 10;

iDigitsAfterDecimal--;

}

frac = new Fraction((int)Math.Round(dTemp), iMultiple);

}

return frac;

}

}

catch (OverflowException)

{

throw new FractionException("Conversion not possible due to overflow");

}

catch (Exception)

{

throw new FractionException("Conversion not possible");

}

}

/// <summary>

/// The function replicates current Fraction object

/// </summary>

public Fraction Duplicate()

{

Fraction frac = new Fraction();

frac.Numerator = Numerator;

frac.Denominator = Denominator;

return frac;

}

/// <summary>

/// The function returns the inverse of a Fraction object

/// </summary>

public static Fraction Inverse(Fraction frac1)

{

if (frac1.Numerator == 0)

throw new FractionException("Operation not possible (Denominator cannot be assigned a ZERO Value)");

long iNumerator = frac1.Denominator;

long iDenominator = frac1.Numerator;

return (new Fraction(iNumerator, iDenominator));

}

/// <summary>

/// Operators for the Fraction object

/// includes -(unary), and binary opertors such as +,-,\*,/

/// also includes relational and logical operators such as ==,!=,<,>,<=,>=

/// </summary>

public static Fraction operator -(Fraction frac1)

{ return (Negate(frac1)); }

public static Fraction operator +(Fraction frac1, Fraction frac2)

{ return (Add(frac1, frac2)); }

public static Fraction operator +(int iNo, Fraction frac1)

{ return (Add(frac1, new Fraction(iNo))); }

public static Fraction operator +(Fraction frac1, int iNo)

{ return (Add(frac1, new Fraction(iNo))); }

public static Fraction operator +(double dbl, Fraction frac1)

{ return (Add(frac1, Fraction.ToFraction(dbl))); }

public static Fraction operator +(Fraction frac1, double dbl)

{ return (Add(frac1, Fraction.ToFraction(dbl))); }

public static Fraction operator -(Fraction frac1, Fraction frac2)

{ return (Add(frac1, -frac2)); }

public static Fraction operator -(int iNo, Fraction frac1)

{ return (Add(-frac1, new Fraction(iNo))); }

public static Fraction operator -(Fraction frac1, int iNo)

{ return (Add(frac1, -(new Fraction(iNo)))); }

public static Fraction operator -(double dbl, Fraction frac1)

{ return (Add(-frac1, Fraction.ToFraction(dbl))); }

public static Fraction operator -(Fraction frac1, double dbl)

{ return (Add(frac1, -Fraction.ToFraction(dbl))); }

public static Fraction operator \*(Fraction frac1, Fraction frac2)

{ return (Multiply(frac1, frac2)); }

public static Fraction operator \*(int iNo, Fraction frac1)

{ return (Multiply(frac1, new Fraction(iNo))); }

public static Fraction operator \*(Fraction frac1, int iNo)

{ return (Multiply(frac1, new Fraction(iNo))); }

public static Fraction operator \*(double dbl, Fraction frac1)

{ return (Multiply(frac1, Fraction.ToFraction(dbl))); }

public static Fraction operator \*(Fraction frac1, double dbl)

{ return (Multiply(frac1, Fraction.ToFraction(dbl))); }

public static Fraction operator /(Fraction frac1, Fraction frac2)

{ return (Multiply(frac1, Inverse(frac2))); }

public static Fraction operator /(int iNo, Fraction frac1)

{ return (Multiply(Inverse(frac1), new Fraction(iNo))); }

public static Fraction operator /(Fraction frac1, int iNo)

{ return (Multiply(frac1, Inverse(new Fraction(iNo)))); }

public static Fraction operator /(double dbl, Fraction frac1)

{ return (Multiply(Inverse(frac1), Fraction.ToFraction(dbl))); }

public static Fraction operator /(Fraction frac1, double dbl)

{ return (Multiply(frac1, Fraction.Inverse(Fraction.ToFraction(dbl)))); }

public static bool operator ==(Fraction frac1, Fraction frac2)

{ return frac1.Equals(frac2); }

public static bool operator !=(Fraction frac1, Fraction frac2)

{ return (!frac1.Equals(frac2)); }

public static bool operator ==(Fraction frac1, int iNo)

{ return frac1.Equals(new Fraction(iNo)); }

public static bool operator !=(Fraction frac1, int iNo)

{ return (!frac1.Equals(new Fraction(iNo))); }

public static bool operator ==(Fraction frac1, double dbl)

{ return frac1.Equals(new Fraction(dbl)); }

public static bool operator !=(Fraction frac1, double dbl)

{ return (!frac1.Equals(new Fraction(dbl))); }

public static bool operator <(Fraction frac1, Fraction frac2)

{ return frac1.Numerator \* frac2.Denominator < frac2.Numerator \* frac1.Denominator; }

public static bool operator >(Fraction frac1, Fraction frac2)

{ return frac1.Numerator \* frac2.Denominator > frac2.Numerator \* frac1.Denominator; }

public static bool operator <=(Fraction frac1, Fraction frac2)

{ return frac1.Numerator \* frac2.Denominator <= frac2.Numerator \* frac1.Denominator; }

public static bool operator >=(Fraction frac1, Fraction frac2)

{ return frac1.Numerator \* frac2.Denominator >= frac2.Numerator \* frac1.Denominator; }

/// <summary>

/// overloaed user defined conversions: from numeric data types to Fractions

/// </summary>

public static implicit operator Fraction(long lNo)

{ return new Fraction(lNo); }

public static implicit operator Fraction(double dNo)

{ return new Fraction(dNo); }

public static implicit operator Fraction(string strNo)

{ return new Fraction(strNo); }

/// <summary>

/// overloaed user defined conversions: from fractions to double and string

/// </summary>

public static explicit operator double(Fraction frac)

{ return frac.ToDouble(); }

public static implicit operator string(Fraction frac)

{ return frac.ToString(); }

/// <summary>

/// checks whether two fractions are equal

/// </summary>

public override bool Equals(object obj)

{

Fraction frac = (Fraction)obj;

return (Numerator == frac.Numerator && Denominator == frac.Denominator);

}

/// <summary>

/// returns a hash code for this fraction

/// </summary>

public override int GetHashCode()

{

return (Convert.ToInt32((Numerator ^ Denominator) & 0xFFFFFFFF));

}

/// <summary>

/// internal function for negation

/// </summary>

private static Fraction Negate(Fraction frac1)

{

long iNumerator = -frac1.Numerator;

long iDenominator = frac1.Denominator;

return (new Fraction(iNumerator, iDenominator));

}

/// <summary>

/// internal functions for binary operations

/// </summary>

public static Fraction Add(Fraction frac1, Fraction frac2)

{

try

{

checked

{

long iNumerator = frac1.Numerator \* frac2.Denominator + frac2.Numerator \* frac1.Denominator;

long iDenominator = frac1.Denominator \* frac2.Denominator;

return (new Fraction(iNumerator, iDenominator));

}

}

catch (OverflowException)

{

throw new FractionException("Overflow occurred while performing arithemetic operation");

}

catch (Exception)

{

throw new FractionException("An error occurred while performing arithemetic operation");

}

}

private static Fraction Multiply(Fraction frac1, Fraction frac2)

{

try

{

checked

{

long iNumerator = frac1.Numerator \* frac2.Numerator;

long iDenominator = frac1.Denominator \* frac2.Denominator;

return (new Fraction(iNumerator, iDenominator));

}

}

catch (OverflowException)

{

throw new FractionException("Overflow occurred while performing arithemetic operation");

}

catch (Exception)

{

throw new FractionException("An error occurred while performing arithemetic operation");

}

}

/// <summary>

/// The function returns GCD of two numbers (used for reducing a Fraction)

/// </summary>

private static long GCD(long iNo1, long iNo2)

{

// take absolute values

if (iNo1 < 0) iNo1 = -iNo1;

if (iNo2 < 0) iNo2 = -iNo2;

do

{

if (iNo1 < iNo2)

{

long tmp = iNo1; // swap the two operands

iNo1 = iNo2;

iNo2 = tmp;

}

iNo1 = iNo1 % iNo2;

} while (iNo1 != 0);

return iNo2;

}

/// <summary>

/// The function reduces(simplifies) a Fraction object by dividing both its numerator

/// and denominator by their GCD

/// </summary>

public static void ReduceFraction(Fraction frac)

{

try

{

if (frac.Numerator == 0)

{

frac.Denominator = 1;

return;

}

long iGCD = GCD(frac.Numerator, frac.Denominator);

frac.Numerator /= iGCD;

frac.Denominator /= iGCD;

if (frac.Denominator < 0) // if -ve sign in denominator

{

//pass -ve sign to numerator

frac.Numerator \*= -1;

frac.Denominator \*= -1;

}

} // end try

catch (Exception exp)

{

throw new FractionException("Cannot reduce Fraction: " + exp.Message);

}

}

} //end class Fraction

/// <summary>

/// Exception class for Fraction, derived from System.Exception

/// </summary>

public class FractionException : Exception

{

public FractionException() : base()

{ }

public FractionException(string Message) : base(Message)

{ }

public FractionException(string Message, Exception InnerException) : base(Message, InnerException)

{ }

} //end class FractionException

public static string SumFracts(int[,] l)

{

// your code

if (l.Length == 0) return null;

Fraction sum = new Fraction(l[0, 0], l[0, 1]);

for (int i = 1; i < l.GetLength(0); i++)

{

if (l.GetLength(1) > 0)

{

sum += new Fraction(l[i, 0], l[i, 1]);

}

else

{

sum += new Fraction(l[i, 0],1);

}

}

return sum.ToString();

}

static void Main()

{

//Fraction f1 = new Fraction(5, 6);

//Fraction f2 = new Fraction(9, 4);

//Fraction res = Fraction.Add(f1, f2);

//Console.WriteLine(res.ToString());

//int[,] a = new int[,] { { 1, 2 }, { 2, 9 }, { 3, 18 }, { 4, 24 }, { 6, 48 } };

//String r = "[85, 72]";

// int[,] a = new int[,] { { 1, 2 }, { 1, 3 }, { 1, 4 } };

int[,] a = new int[,] { };

Console.WriteLine(SumFracts(a));

Console.ReadLine();

}

}

**using System;**

**using System.Collections.Generic;**

**using System.Linq;**

**public static class SumFractions**

**{**

**private static int Gcd(int a, int b) => b == 0 ? a : Gcd(b, a % b);**

**private static int Lcm(int a, int b) => (a \* b) / Gcd(a, b);**

**private static int Lcm(IEnumerable<int> numbers) => numbers.Aggregate(Lcm);**

**private static bool Divides(this int a, int b) => b % a == 0;**

**public static string SumFracts(int[,] fractions)**

**{**

**if (fractions.Length == 0) return null;**

**var tuples = Enumerable**

**.Range(0, fractions.GetLength(0))**

**.Select(i => Tuple.Create(fractions[i, 0], fractions[i, 1]))**

**.ToArray();**

**var denominator = Lcm(tuples.Select(t => t.Item2));**

**var numerator = tuples.Select(t => t.Item1 \* (denominator / t.Item2)).Sum();**

**int gcd;**

**while ((gcd = Gcd(numerator, denominator)) > 1)**

**{**

**numerator /= gcd;**

**denominator /= gcd;**

**}**

**return denominator == 1**

**? $"{numerator}"**

**: $"[{numerator}, {denominator}]";**

**}**

**}**