main.rs

```rust
mod entier;
mod rationnel;

use rationnel::*;
use entier::est_premier;

fn main() {
  println!("Somme = {}", (1..5+1).fold(10, |somme, element| somme+element));
  for i in (1..19).filter(|n| n%2==1 ) { print!("{} ", i)} println!();
  for i in (0..9).map(|n| 2*n+1 ) { print!("{} ", i)} println!();

  for entier in 2..110 {
    print!("{:#.21}", Rationnel::new(1, entier));
    println!(" = {:+2.3}", Rationnel::new(1, entier));
  }

  let premiers = (2..52).filter(|n| est_premier(*n)).collect::<Vec<_>>();
  println!("Premiers {:?}", premiers);
  for window in premiers.windows(2) {
    let (debut, fin) = (window[0], window[window.len()-1]);
    let r1 = Rationnel::new(1, debut);
    let r2 = Rationnel::new(1, fin);
    print!("{:.2} - {:.2}", &r1, &r2);
    let r3 = r1-r2;
    println!(" = {:+#11.18}", r3);
  }
}
```

entier.rs

```rust
use std::fmt::{Display, Formatter, Result};
use NombreEntier::{Diviseurs, Premier};
use std::ops::{Sub, Mul};

#[derive(PartialEq, Clone)]
pub enum NombreEntier {
  Premier(usize),
  Diviseurs(usize, Vec<usize>)
}

impl NombreEntier {
  /// # Méthode publique statique new()
  /// * Création d'un entier naturel qui permet de connaître si le nombre est premier
  /// ou s'il possède des diviseurs.
  /// # Paramètre :
  /// * `nombre`: entier naturel enregistré dans l'énumération
  /// * returns: NombreEntier
  pub fn new(nombre: usize) -> NombreEntier {
    let premiers = liste_diviseurs(nombre);
    if premiers.len()==1 { Premier(nombre) } else { Diviseurs(nombre, premiers) }
  }
  /// # Méthode publique nombre()
  /// Retourne la valeur numérique du nombre entier naturel sauvegardé qu'il soit premier
  /// ou qu'il comporte des diviseurs
  pub fn nombre(self) -> usize {
    match self {
      Premier(x) => x,
      Diviseurs(x, _) => x
    }
  }
}

impl Sub for NombreEntier {
  type Output = NombreEntier;
  /// # Méthode sub() :
  /// * Redéfinition de l'opérateur '-' qui permet de réaliser la soustraction naturellement entre deux nombres
entiers.
  /// # Paramètre :
  /// * `autre`: Il s'agit du deuxième nombre entier
  /// * returns: <NombreEntier as Mul<Self>>::Output : Nouveau nombre entier correspondant au résultat
  /// du calcul.
  /// # Exemples :
  /// >       let x = NombreEntier::new(7);
  /// >       let y = NombreEntier::new(24);
  /// >       let z = y - x;
  /// # Valeur de z :
  /// >       17 [premier]
```

```rust
  fn sub(self, autre: Self) -> Self::Output {
    let n = self.nombre() - autre.nombre();
    let premiers = liste_diviseurs(n);
    if premiers.len()==1 { Premier(n) } else { Diviseurs(n, premiers) }
  }
}

impl Mul for NombreEntier {
  type Output = NombreEntier;

  /// # Méthode mul() :
  /// * Redéfinition de l'opérateur '*' qui permet de réaliser la multiplication naturellement entre deux nombres entiers.
  /// # Paramètre :
  /// * `autre`: Il s'agit du deuxième nombre entier
  /// * returns: <NombreEntier as Mul<Self>>::Output : Nouveau nombre entier correspondant au résultat
  /// du calcul.
  /// # Exemples :
  /// >       let x = NombreEntier::new(7);
  /// >       let y = NombreEntier::new(24);
  /// >       let z = x * y;
  /// # Valeur de z :
  /// >       168 [2, 2, 2, 3, 7]
  fn mul(self, autre: Self) -> Self::Output {
    let n = self.nombre() * autre.nombre();
    let premiers = liste_diviseurs(n);
    if premiers.len()==1 { Premier(n) } else { Diviseurs(n, premiers) }
  }
}

impl Display for NombreEntier {
  /// # Méthode fmt() :
  /// Propose un affichage circonstancié du nombre entier
  /// * soit uniquement la valeur du nombre
  /// * soit la valeur du nombre suivi du qualificatif : nombre premier ou liste des diviseurs
  /// # Paramètre :
  /// * `f`: maîtrise le formatage de la chaîne de caractères.
  /// Prise en compte des justifications à droite ou à gauche, ainsi que la largeur
  /// et la précision souhaitées grâce à la méthode pad().
  /// Le choix de l'affichage complet ou pas se fait au travers de la méthode alternate().
  /// Cette dernière méthode est active lorsque nous proposons le symbole '#'.
  /// # Exemple :
  /// * format!("{:.>#20}", 18) : demande un affichage complet avec une justification à droite et un gabarit de 15 caractères
  /// # Résultat :
  /// >.........18 [2, 3, 3]
  fn fmt(&self, f: &mut Formatter<'_>) -> Result {
    let motif = format!("{}", if f.alternate() {
      match self {
        Premier(nombre) => format!("{} [premier]", nombre),
        Diviseurs(nombre, liste_diviseurs) => format!("{} {:?}", nombre, liste_diviseurs)
      }
    }
    else { match self {
        Premier(nombre) => format!("{}", nombre),
        Diviseurs(nombre, _) => format!("{}", nombre)
      }
    });
    f.pad(motif.as_str())
  }
}
/// # Fonction publique liste_diviseur()
/// * Renvoi la liste des nombres premiers à partir d'un nombre entier naturel en argument.
/// # Paramètre :
/// * nombre: entier naturel qui sert pour le traitement de la fonction
/// * retour : Vec<usize, Global> : l'ensemble des nombres premiers est placé dans un vecteur
pub fn liste_diviseurs(mut nombre: usize) -> Vec<usize> {
  let mut premiers = Vec::new();
  loop {
    let mut trouve = false;
    let borne = f64::sqrt(nombre as f64) as usize;
    for diviseur in 2..borne+1 {
      if nombre % diviseur == 0 {
        premiers.push(diviseur);
        nombre /= diviseur;
        trouve = true;
        break;
      }
```

```
    }
    if !trouve { break }
  }
  premiers.push(nombre);
  premiers
}

/// # Fonction publique est_premier() :
/// * Cette fonction permet de déterminer si un nombre entier naturel est un nombre premier
/// # Paramètre :
/// * `n` : le paramètre est le nombre naturel qui sert à l'évaluation
/// * returns : bool (est premier ou possède des diviseurs)
pub fn est_premier(n: usize) -> bool {
  let borne = f64::sqrt(n as f64) as usize;
  for diviseur in 2..borne { if n % diviseur == 0 { return false }  }
  true
}
```

rationnel.rs

```
use std::fmt::{Display, Formatter, Result};
use crate::entier::{NombreEntier, NombreEntier::Diviseurs, NombreEntier::Premier};
use std::ops::Sub;

#[derive(PartialEq, Clone)]
pub struct Rationnel {
  numerateur: NombreEntier,
  denominateur: NombreEntier
}

impl Rationnel {
  /// # Méthode publique statique new()
  /// Création d'un nombre rationnel à partir de deux nombres entiers naturels.
  /// Durant la création, le rationnel est automatiquement simplifié s'il existe des
  /// diviseurs communs à la fois sur le numérateur et le dénominateur.
  /// # Paramètres :
  /// * `numerateur`: entier naturel
  /// * `denominateur`: entier naturel
  /// * retour: Rationnel : le nombre rationnel simplifié
  /// # Exemple :
  /// >       let r = Rationnel::new(6, 21);
  /// # Résultat :
  /// >       r = 2/7
  pub fn new(numerateur: usize, denominateur: usize) -> Rationnel {
    let mut rationnel = Rationnel {
      numerateur: NombreEntier::new(numerateur),
      denominateur: NombreEntier::new(denominateur)
    };
    rationnel.simplication();
    rationnel
  }
  /// # Méthode privée simplification()
  /// Utilisée uniquement pour simplifier le nombre rationnel lors de sa création.
  /// Aucune utilisation pour l'instant en dehors de cette phase de création.
  fn simplication(&mut self) {
    let numerateur = match self.numerateur.clone() {
      Premier(n) => vec![n],
      Diviseurs(_, diviseurs) => diviseurs
    };
    let mut denominateur = match self.denominateur.clone() {
      Premier(n) => vec![n],
      Diviseurs(_, diviseurs) => diviseurs
    };
    let mut num = 1;
    for n in numerateur {
      if denominateur.contains(&n) {
        let index = denominateur.iter().position(|x| x == &n).unwrap();
        denominateur.remove(index);
      }
      else { num *= n }
    }
    let mut deno = 1;
    for den in denominateur { deno *= den  }
    self.numerateur = NombreEntier::new(num);
    self.denominateur = NombreEntier::new(deno);
  }
  /// # Méthode privée decimales()
  /// Utilisée uniquement pour la redéfinition du trait Display. C'est cette méthode qui permet
  /// d'avoir un affichage personnalisé.
```

```rust
  /// # Paramètres :
  /// * `gauche`: gabarit pour la justification à gauche du numérateur
  /// * `droite`: gabarit pour la justification à droite du dénominateur
  /// * `division`: affiche ou pas la valeur décimale équivalente du rationnel
  /// * `complet`: affichage complet ou réduit du nombre rationnel
  /// * retour: String : la chaîne complète en correspondance avec l'affichage demandé.
  fn decimales(&self, gauche: usize, droite: usize, division: bool, complet: bool) -> String {
    if division {
      let calcul = self.retrouve_decimales();
      let (mut periode, decalage, chiffres) = calcul_periodicite(&calcul);
      let juste = chiffres.len()==1 && chiffres[0]==0;
      if juste { periode=0 };
      let mut decimales = String::new();
      for n in &calcul[..decalage] { decimales.push_str(format!("{}", n).as_str()) }
      if juste { decimales.push_str(" (juste)")}
      else { decimales.push_str("[");
        for n in &chiffres { decimales.push_str(format!("{}", n).as_str()); }
        decimales.push_str("]");
      };
      if complet { format!("{:>#gauche$} / {:<#droite$} = p={:<5}0,{}", self.numerateur, self.denominateur,
periode, decimales) }
      else { format!("{:>gauche$} / {:<droite$} = p={:<5}0,{}", self.numerateur, self.denominateur, periode,
decimales) }

    }
    else {
      if complet { format!("{:>#gauche$} / {:<#droite$}", self.numerateur, self.denominateur) }
      else { format!("{:>gauche$} / {:<droite$}", self.numerateur, self.denominateur) }
    }
  }
  /// # Méthode privée retrouve_decimales() :
  /// Retourne la liste des décimales du nombre rationnel. Cette liste est sous forme d'un vecteur
  /// dont la longueur correspond au double de la valeur du dénominateur.
  fn retrouve_decimales(&self) -> Vec<usize> {
    let diviseur = self.denominateur.clone().nombre();
    let mut dividende = self.numerateur.clone().nombre();
    dividende *= 10;
    let mut decimales= vec![];
    let maxi = diviseur*2;
    for _ in 0..maxi {
      decimales.push(dividende / diviseur);
      dividende = (dividende % diviseur) * 10;
    }
    decimales
  }
}

impl Sub for Rationnel {
  type Output = Rationnel;

  /// # Méthode sub() :
  /// Redéfinition de cette méthode qui permet de réaliser une soustraction entre deux nombres
  /// rationnels directement à l'aide du symbole '-'.
  /// # Paramètre :
  /// * `autre`: deuxième nombre rationnel qui permet d'effectuer la soustraction.
  /// * returns: <Rationnel as Sub<Self>>::Output : Résultat de la soustraction avec la création
  /// d'un nouveau nombre rationnel.
  /// # Exemple :
  /// >       let x = Rationnel::new(1, 2);
  /// >       let y = Rationnel::new(1, 3);
  /// >       let z = x - y;
  /// # Résultat :
  /// >       z = 1/6
  fn sub(self, autre: Self) -> Self::Output {
    let mut rationnel = Rationnel {
      numerateur : self.numerateur*autre.denominateur.clone() - autre.numerateur*self.denominateur.clone(),
      denominateur: self.denominateur * autre.denominateur
    };
    rationnel.simplication();
    rationnel
  }
}

impl Display for Rationnel {
  /// # Méthode fmt() :
  /// Propose un affichage paramétré du nombre rationnel en tenant compte des critères d'affichage
  /// de la part de l'utilisateur :
  /// * justification à droite du numérateur,
```

```rust
  /// * justification à gauche du dénominateur,
  /// * résultat de la division ou pas.
  /// * spécification ou pas de l'état premier ou de la liste des divisuers pour le numérateur et le dénominateur.
  /// # Paramètre :
  /// * `f`: maîtrise le formatage de la chaîne de caractères suivant les critères spécifiés ci-dessus.
  /// * méthode width() correspond à la justification à droite (numérateur),
  /// * méthode precision() indique la justification à gauche (dénominateur),
  /// * méthode sign_plus() '+' demande l'affichage du résultat de la division avec la périodicité,
  /// * méthode alternate() '#' impose la nature du numérateur et du dénominateur savoir premier ou liste des
diviseurs
  /// # Exemple :
  /// >       let x = Rationnel::new(1, 6);
  /// >       println!("{:+#11.18}", x);
  /// # Résultat :
  /// >       1 [premier] / 6 [2, 3]          =  1 / 6   = p=1    0,1[6]
  fn fmt(&self, f: &mut Formatter<'_>) -> Result {
    let gauche = f.width().unwrap_or(1);
    let droite = f.precision().unwrap_or(1);
    let division = f.sign_plus();
    let complet = f.alternate();
    write!(f, "{}", self.decimales(gauche, droite, division, complet))
  }
}

/// # Fonction privée calcul_periodicite() :
/// Permet de retrouver la périodicité, du décalage de cette périodicité suivi de l'ensemble
/// des chiffres composant cette périodicité à partir de l'ensemble des décimales issues du
/// calcul de la division entière.
/// # Paramètre :
/// * `decimales`: la liste des chiffres composant le calcul de la division sous forme de vecteur
///
/// * retour : (usize, usize, Vec<usize, Global>) : Dans l'ordre : la périodicité, le décalage
/// et la liste des chiffres de la périodicité.
fn calcul_periodicite(decimales: &Vec<usize>) -> (usize, usize, Vec<usize>) {
  let (taille, mut chiffres) = (decimales.len(), vec![]);
  let (mut periode, mut decalage, mut periodique) = (1, 0, false);
  while !periodique {
    while decalage+periode < taille && !(decimales[decalage+periode] == decimales[decalage]) { periode+=1; }
    if decalage+periode == taille { periodique=false; }
    else {
      periodique=true;
      for i in decalage..decalage+periode {
        if i+periode == taille || (decimales[i]!=decimales[i+periode]) { periodique=false; break;  }
      }
    }
    if periodique && periode==1 {
      for i in decalage+periode..taille/2 {
        if decimales[i]!=decimales[i+periode] { periodique=false; break;  }
      }
    }
    if !periodique {
      if (decalage+periode)==taille { decalage+=1; periode=1;  }
      else { periode+=1;  continue;  }
    }
    if (decalage+periode)>=taille { break; }
    else { chiffres = decimales.as_slice()[decalage..decalage+periode].to_vec() }
  }
  (periode, decalage, chiffres)
}
```

Résultats

```
  Somme = 25
1 3 5 7 9 11 13 15 17
1 3 5 7 9 11 13 15 17
1 [premier] / 2 [premier]          =  1 / 2   = p=0    0,5 (juste)
1 [premier] / 3 [premier]          =  1 / 3   = p=1    0,[3]
1 [premier] / 4 [2, 2]             =  1 / 4   = p=0    0,25 (juste)
1 [premier] / 5 [premier]          =  1 / 5   = p=0    0,2 (juste)
1 [premier] / 6 [2, 3]             =  1 / 6   = p=1    0,1[6]
1 [premier] / 7 [premier]          =  1 / 7   = p=6    0,[142857]
1 [premier] / 8 [2, 2, 2]          =  1 / 8   = p=0    0,125 (juste)
1 [premier] / 9 [3, 3]             =  1 / 9   = p=1    0,[1]
1 [premier] / 10 [2, 5]            =  1 / 10  = p=0    0,1 (juste)
1 [premier] / 11 [premier]         =  1 / 11  = p=2    0,[09]
1 [premier] / 12 [2, 2, 3]         =  1 / 12  = p=1    0,08[3]
1 [premier] / 13 [premier]         =  1 / 13  = p=6    0,[076923]
1 [premier] / 14 [2, 7]            =  1 / 14  = p=6    0,0[714285]
1 [premier] / 15 [3, 5]            =  1 / 15  = p=1    0,0[6]
```

```
1 [premier] / 16 [2, 2, 2, 2]      =  1 / 16  = p=0    0,0625 (juste)
1 [premier] / 17 [premier]         =  1 / 17  = p=16   0,[0588235294117647]
1 [premier] / 18 [2, 3, 3]         =  1 / 18  = p=1    0,0[5]
1 [premier] / 19 [premier]         =  1 / 19  = p=18   0,[052631578947368421]
1 [premier] / 20 [2, 2, 5]         =  1 / 20  = p=0    0,05 (juste)
1 [premier] / 21 [3, 7]            =  1 / 21  = p=6    0,[047619]
1 [premier] / 22 [2, 11]           =  1 / 22  = p=2    0,0[45]
1 [premier] / 23 [premier]         =  1 / 23  = p=22   0,[0434782608695652173913]
1 [premier] / 24 [2, 2, 2, 3]      =  1 / 24  = p=1    0,041[6]
1 [premier] / 25 [5, 5]            =  1 / 25  = p=0    0,04 (juste)
1 [premier] / 26 [2, 13]           =  1 / 26  = p=6    0,0[384615]
1 [premier] / 27 [3, 3, 3]         =  1 / 27  = p=3    0,[037]
1 [premier] / 28 [2, 2, 7]         =  1 / 28  = p=6    0,03[571428]
1 [premier] / 29 [premier]         =  1 / 29  = p=28   0,[0344827586206896551724137931]
1 [premier] / 30 [2, 3, 5]         =  1 / 30  = p=1    0,0[3]
1 [premier] / 31 [premier]         =  1 / 31  = p=15   0,[032258064516129]
1 [premier] / 32 [2, 2, 2, 2, 2]   =  1 / 32  = p=0    0,03125 (juste)
1 [premier] / 33 [3, 11]           =  1 / 33  = p=2    0,[03]
1 [premier] / 34 [2, 17]           =  1 / 34  = p=16   0,0[2941176470588235]
1 [premier] / 35 [5, 7]            =  1 / 35  = p=6    0,0[285714]
1 [premier] / 36 [2, 2, 3, 3]      =  1 / 36  = p=1    0,02[7]
1 [premier] / 37 [premier]         =  1 / 37  = p=3    0,[027]
1 [premier] / 38 [2, 19]           =  1 / 38  = p=18   0,0[263157894736842105]
1 [premier] / 39 [3, 13]           =  1 / 39  = p=6    0,[025641]
1 [premier] / 40 [2, 2, 2, 5]      =  1 / 40  = p=0    0,025 (juste)
1 [premier] / 41 [premier]         =  1 / 41  = p=5    0,[02439]
1 [premier] / 42 [2, 3, 7]         =  1 / 42  = p=6    0,0[238095]
Premiers [2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 15, 17]
1 / 2  - 1 / 3  = 1 [premier] / 6 [2, 3]          = p=1    0,1[6]
1 / 3  - 1 / 4  = 1 [premier] / 12 [2, 2, 3]      = p=1    0,08[3]
1 / 4  - 1 / 5  = 1 [premier] / 20 [2, 2, 5]      = p=0    0,05 (juste)
1 / 5  - 1 / 6  = 1 [premier] / 30 [2, 3, 5]      = p=1    0,0[3]
1 / 6  - 1 / 7  = 1 [premier] / 42 [2, 3, 7]      = p=6    0,0[238095]
1 / 7  - 1 / 8  = 1 [premier] / 56 [2, 2, 2, 7]   = p=6    0,017[857142]
1 / 8  - 1 / 9  = 1 [premier] / 72 [2, 2, 2, 3, 3] = p=1    0,013[8]
1 / 9  - 1 / 11 = 2 [premier] / 99 [3, 3, 11]     = p=2    0,[02]
1 / 11 - 1 / 13 = 2 [premier] / 143 [11, 13]      = p=6    0,[013986]
1 / 13 - 1 / 15 = 2 [premier] / 195 [3, 5, 13]    = p=6    0,0[102564]
1 / 15 - 1 / 17 = 2 [premier] / 255 [3, 5, 17]    = p=16   0,0[0784313725490196]
```