Lesson 1: Continued fractions

Created: 2024-05-27T01:25:57.339493+10:00

Continued fractions

Continued fractions

Continued fractions

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming

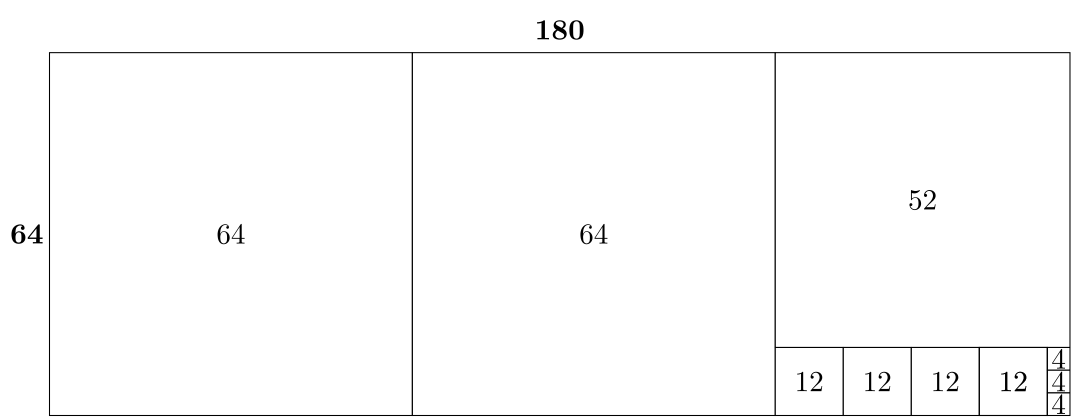
[1]:
```python
import math
from itertools import cycle, chain, repeat
from fractions import Fraction
```

0.0.1 Paving a rectangle by squares, Euclid's algorithm for computing the greatest common divisor, and finite continued fractions

Euclid's algorithm determines that $\gcd(180, 64) = 4$ by performing the computations displayed in red in the following:

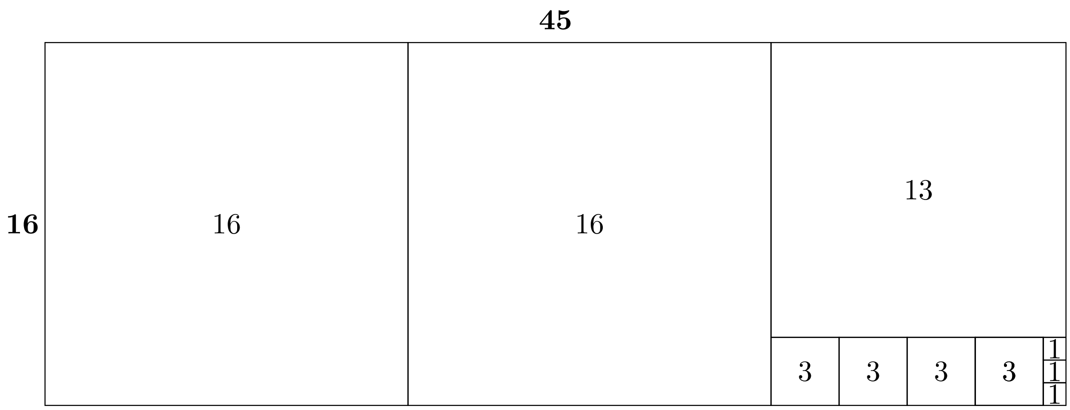
```
180    =    180 // 64 * 64 + 180 % 64    =    2 * 64 + 52
64     =     64 // 52 * 52 + 64  % 52    =    1 * 52 + 12
52     =     52 // 12 * 12 + 52  % 12    =    4 * 12 +  4
12     =     12 //  4 *  4 + 12  %  4    =    3 * 12 +  0
```

It corresponds to finding out that 4 is the size of the largest square thanks to which it is possible to pave a rectangle of size 180 by 64, based on the following geometric construction:



So when the gcd is 1, the paving of the rectangle can only be achieved with squares of size 1 by 1:

```
45    =    45  // 16 * 16 + 45 % 16    =    2 * 16 + 13
16    =    16  // 13 * 13 + 16 % 13    =    1 * 13 +  3
13    =    13  //  3 *  3 + 13 %  3    =    4 *  3 +  1
 3    =     3  //  1 *  3  + 3 %  1    =    3 *  1 +  0
```



The blue part on the right hand sides of both previous sets of equations is the same, and the pictures illustrate that

$$\frac{180}{64} = \frac{45}{16} = 2 + \cfrac{1}{1 + \cfrac{1}{4 + \frac{1}{3}}}$$

corresponding to the fact that $\frac{180}{64} = \frac{45}{16} = 2 + \frac{13}{16} = 2 + \frac{13}{13+3} = 2 + \frac{1}{1+\frac{3}{13}} = 2 + \frac{1}{1+\frac{3}{12+1}} = 2 + \frac{1}{1+\frac{3}{4+\frac{1}{3}}}$.

The pictures illustrate that more generally, any rational number can be written as:

$$a_0 + 1/(a_1 + 1/(a_2 + \cdots + 1/a_k \overbrace{)\ldots)}^{k}$$

where $a_0 \in \mathbf{Z}$, $k \in \mathbf{N}$, and $a_1, \ldots, a_k \in \mathbf{N} \setminus \{0\}$ with $a_k \neq 1$, which is the general form of a *finite continued fraction*, that it is convenient to denote by $[a_0, a_1, a_2, \ldots, a_k]$. Note that we could allow a finite continued fraction to end in 1 because for all $b \in \mathbf{N} \setminus \{0, 1\}$, $b = b - 1 + \frac{1}{1}$; that would make $[a_0, a_1, a_2, \ldots, a_k - 1, 1]$ an alternative representation to $[a_0, a_1, a_2, \ldots, a_k]$.

0.0.2 Computation of a finite continued fraction

More generally, given $k \in \mathbf{N}$ and $a_0, \ldots, a_k \in \mathbf{Z}$ with $a_0, \ldots, a_k$ at least equal to 1, let $[a_0, \ldots, a_k]$ be defined as a_0 if $k = 0$, and as $a_0 + \frac{1}{[a_1, \ldots, a_k]}$ if $k > 0$. For all $i \in \{-2, \ldots, k\}$:

- let p_i be equal to 0 if $i = -2$, to 1 if $i = -1$, and to $a_i p_{i-1} + p_{i-2}$ if $i \geq 0$;
- let q_i be equal to 1 if $i = -2$, to 0 if $i = -1$, and to $a_i q_{i-1} + q_{i-2}$ if $i \geq 0$.

A trivial proof by induction shows that for all $j \in \{0, \ldots, k\}$, $q_j > 0$. We now show that:

$$[a_0, \ldots, a_k] = \frac{p_k}{q_k} \tag{1}$$

which provides an effective method for computing $[a_0, \ldots, a_k]$.

Towards proving (1), first define for all $j \in \{-1, \ldots, k\}$ the matrix M_j as

$$\begin{bmatrix} p_j & q_j \\ p_{j-1} & q_{j-1} \end{bmatrix}$$

It is immediately verified by induction that for all $j \in \{0, \dots, k\}$,

$$M_j = \begin{bmatrix} a_j & 1 \\ 1 & 0 \end{bmatrix} M_{j-1},$$

from which it follows that for all $j \in \{0, \dots, k\}$,

$$\begin{bmatrix} p_j & q_j \\ p_{j-1} & q_{j-1} \end{bmatrix} = \begin{bmatrix} a_j & 1 \\ 1 & 0 \end{bmatrix} \dots \begin{bmatrix} a_0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2}$$

As the transpose of the product of two matrixes A and B is the product of the transpose of B by the transpose of A, we have that for all $j \in \{0, \dots, k\}$,

$$\begin{bmatrix} p_j & p_{j-1} \\ q_j & q_{j-1} \end{bmatrix} = \begin{bmatrix} a_0 & 1 \\ 1 & 0 \end{bmatrix} \dots \begin{bmatrix} a_j & 1 \\ 1 & 0 \end{bmatrix}$$

which implies that for all $j \in \{0, \dots, k\}$,

$$\begin{bmatrix} p_j \\ q_j \end{bmatrix} = \begin{bmatrix} a_0 & 1 \\ 1 & 0 \end{bmatrix} \dots \begin{bmatrix} a_j & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{3}$$

Now proof of (1) is by induction on the length of finite continued fractions and application of (3). It is trivial that if $k = 0$ then (1) holds. Assume that $k > 0$. Denoting $[a_1, \dots, a_k]$ by $\frac{u}{v}$, we have by induction and (3) that

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} a_1 & 1 \\ 1 & 0 \end{bmatrix} \dots \begin{bmatrix} a_k & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Then $[a_0, \dots, a_k] = a_0 + \frac{1}{[a_1, \dots, a_k]} = a_0 + \frac{v}{u} = \frac{ua_0 + v}{u}$. Hence

$$\begin{bmatrix} p_j \\ q_j \end{bmatrix} = \begin{bmatrix} a_0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} ua_0 + v \\ u \end{bmatrix}$$

Hence $[a_0, \dots, a_k] = \frac{p_j}{q_j}$, which completes the proof of (1).

0.0.3 Infinite continued fractions

Extend the notation of the previous section with $c_j = \frac{p_j}{q_j}$ for all strictly positive $j \leq k$. Then for all $j \in \{2, \dots, k\}$, $c_j - c_{j-1}$ is equal to $\frac{p_j q_{j-1} - p_{j-1} q_j}{q_j q_{j-1}}$. Note that for all $j \leq k$, $p_j q_{j-1} - p_{j-1} q_j$ is the determinant of the matrix M_j, and it then follows from (2) that it is equal to $(-1)^j$. Hence for all strictly positive $j \leq k$,

$$c_j - c_{j-1} = \frac{(-1)^j}{q_j q_{j-1}} \tag{4}$$

Moreover, it is immediately verified by induction that $(q_j)_{2\leq j\leq k}$ is a strictly increasing sequence. This shows that given $a_0 \in \mathbf{Z}$ and a sequence $(a_j)_{j\in\mathbf{N}\setminus\{0\}}$ of members of $\mathbf{N}\setminus\{0\}$, the sequence $([a_0,\dots,a_j])_{j\in\mathbf{N}}$ converges; it is called an *infinite continued fraction* and it is denoted either as

$$a_0 + \cfrac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3+\dots}}}$$

or as $[a_0, a_1, a_2, a_3 \dots]$.

It follows from the previous observations that given an infinite continued fraction $[a_0, a_1, a_2, a_3 \dots]$, $j \in \mathbf{N}\setminus\{0,1\}$ and $n \in \mathbf{N}$, if $[a_0,\dots,a_j]$ and $[a_0,\dots,a_{j+1}]$ agree up to n digits after the decimal point, then $[a_0,\dots,a_j]$ and $[a_0, a_1, a_2, a_3 \dots]$ agree up to n digits after the decimal point. This allows one to compute exactly any approximation of $[a_0, a_1, a_2, a_3 \dots]$.

0.0.4 Negating continued fractions

Given $a \in \mathbf{Z}$, $b \in \mathbf{N}\setminus\{0,1\}$ and $r \in [0,1)$,

$$-\left(a+\frac{1}{b+r}\right) = -a-1+1-\frac{1}{b+r} = -a-1+\frac{b+r-1}{b+r} = -a-1+\frac{1}{\frac{b+r}{b+r-1}} = -a-1+\frac{1}{1+\frac{1}{b-1+r}}$$

Given $a \in \mathbf{Z}$, $c \in \mathbf{N}\setminus\{0\}$ and $r \in [0,1)$,

$$-\left(a+\frac{1}{1+\frac{1}{c+r}}\right) = -a-1+1-\frac{1}{1+\frac{1}{c+r}} = -a-1+\frac{\frac{1}{c+r}}{1+\frac{1}{c+r}} = -a-1+\frac{1}{1+c+r}$$

It follows that, using $\dots$ to denote the possibly missing terms of a finite or infinite continued fraction,

- for all $a \in \mathbf{Z}$ and $b \in \mathbf{N}\setminus\{0,1\}$, $-[a, b\dots] = [-a-1, 1, b-1\dots]$;
- for all $a \in \mathbf{Z}$ and $c \in \mathbf{N}\setminus\{0\}$, $-[a, 1, c\dots] = [-a-1, 1+c\dots]$.

0.0.5 Implementation

Let us define a class `ContinuedFraction` to represent both finite and infinite continued fractions. The `__init()__` function of the class will take two arguments besides `self`, namely, `finite_expansion` and `periodic_expansion`, set to `None` by default, to be possibly modified and become `ContinuedFraction` object attributes.

- When `__init()__` receives no value for `periodic_expansion` or it receives the empty list as a value for `periodic_expansion`, the object will represent a finite continued fraction. The object's `periodic_expansion` attribute will be set to the empty list.
  - In case `__init()__` receives no value for `finite_expansion` or it receives the empty list as a value for `finite_expansion`, the object will represent the finite continued fraction $[0]$, and the object's `finite_expansion` attribute will denote that list.

  - If L is a nonempty list of integers all of which are strictly positive except possibly the first one and `__init()__` receives L as value for `finite_expansion`, then the object will represent the finite continued fraction L. The object's `finite_expansion` attribute will denote L, unless L is of the form $[a_0, \ldots, a_k, 1]$; in that case, the continued fraction is better represented as $[a_0, \ldots, a_k + 1]$ and the object's `finite_expansion` attribute will denote that list rather than L.
- When `__init()__` receives a nonempty list P of strictly positive integers as a value for `periodic_expansion`, the object will represent an infinite continued fraction. Then `__init()__` can receive no value for `finite_expansion`, or it can receive the empty list as a value for `finite_expansion`, or it can receive a nonempty list L of integers all of which are strictly positive except possibly the first one as a value for `finite_expansion`; set L to $[0]$ in the first two cases. The infinite continued fraction that the object represents is meant to be $LP^\star$, that is, the list consisting of the members of L followed by the members of P repeated forever. We simplify and normalise the representation, looking for the shortest list $\widehat{P}$ such that $LP^\star$ can also be written as $L\widehat{P}^\star$, and looking for the shortest list $\overline{L}$ such that $LP^\star$ can also be written as $\overline{L}\,\overline{P}^\star$ for some list $\overline{P}$ of the same length as $\widehat{P}$; then we make $\overline{L}$ and $\overline{P}$ the values of the object's `finite_expansion` and `periodic_expansion` attributes, respectively. For instance:
  - a call to `ContinuedFraction([0, 1], [1])`, `ContinuedFraction([0], [1, 1])`, `ContinuedFraction([0, 1], [1, 1, 1])` or `ContinuedFraction([0, 1, 1], [1, 1, 1])` will create an object that represents the infinite continued fraction $[0, 1, 1, 1...]$, with $[0]$ and $[1]$ as values of the object's `finite_expansion` and `periodic_expansion` attributes, respectively;
  - a call to `ContinuedFraction([0], [1, 2, 1, 2, 1, 2])` will create an object that represents the infinite continued fraction $[0, 1, 2, 1, 2, 1, 2...]$, with $[0]$ and $[1, 2]$ as values of the object's `finite_expansion` and `periodic_expansion` attributes, respectively;
  - a call to `ContinuedFraction([0, 2], [1, 2, 1, 2, 1, 2])` will create an object that represents the infinite continued fraction $[0, 2, 1, 2, 1, 2, 1...]$, with $[0]$ and $[2, 1]$ as values of the object's `finite_expansion` and `periodic_expansion` attributes, respectively;
  - a call to `ContinuedFraction([0, 1, 2, 3], [4, 2, 3, 4, 2, 3])` or `ContinuedFraction([0, 1, 2, 3, 1], [4, 2, 3, 1])` will create an object that represents the infinite continued fraction $[0, 1, 2, 3, 4, 2, 3, 4...]$, with $[0, 1]$ and $[2, 3, 4]$ as values of the object's `finite_expansion` and `periodic_expansion` attributes, respectively.

  $\widehat{P}$ is the shortest list such that P is of the form $\widehat{P} \ldots \widehat{P}$. If L and $\widehat{P}$ end in the same number e, then the last occurrence of e in L can be deleted and the last occurrence of e in $\widehat{P}$ moved to the beginning of $\widehat{P}$, yielding two lists $\widetilde{L}$ and $\widetilde{P}$ such that $LP^\star$ can also be written as $\widetilde{L}\widetilde{P}^*$. Possibly repeated long enough, this transformation eventually yields the desired lists $\overline{L}$ and $\overline{P}$, that end in two distinct numbers.

Putting all this together, we can start the implementation of `ContinuedFraction`, making use a dedicated `Exception` class to deal with incorrect input to `__init()__`. We print out a `ContinuedFraction` object as a single list that represents the finite expansion in case it is for a finite continued fraction, and that represents the finite expansion followed by the periodic expansion, using a semicolon instead of a comma as a separator between both parts, in case it is for an infinite continued fraction:

[2]:
```
class ContinuedFractionError(Exception):
    pass
```

[3]:
```
class ContinuedFraction:
    def __init__(self, finite_expansion=None, periodic_expansion=None):
        if finite_expansion is not None\
           and (not isinstance(finite_expansion, list)
                or any(not isinstance(e, int) for e in finite_expansion)
                or any(e <= 0 for e in finite_expansion[1 :])
               ):
            raise ContinuedFractionError('Incorrect finite expansion')
        if periodic_expansion is not None\
           and (not isinstance(periodic_expansion, list)
                or any(not isinstance(e, int) for e in periodic_expansion)
                or any(e <= 0 for e in periodic_expansion)
               ):
            raise ContinuedFractionError('Incorrect periodic expansion')

        self.finite_expansion = finite_expansion if finite_expansion else [0]
        if periodic_expansion:
            for i in range(1, len(periodic_expansion) // 2 + 1):
                if len(periodic_expansion) % i == 0 and periodic_expansion ==\
                      periodic_expansion[: i] * (len(periodic_expansion) // i):
                    periodic_expansion = periodic_expansion[: i]
                    break
            while len(self.finite_expansion) > 1\
                  and self.finite_expansion[-1] == periodic_expansion[-1]:
                    self.finite_expansion.pop()
                    periodic_expansion.insert(0, periodic_expansion.pop())
            self.periodic_expansion = periodic_expansion
        else:
            self.periodic_expansion = []
            if len(self.finite_expansion) > 1\
               and self.finite_expansion[-1] == 1:
                self.finite_expansion.pop()
                self.finite_expansion[-1] += 1

    def __repr__(self):
        return f'ContinuedFraction({self.finite_expansion}, '\
               f'{self.periodic_expansion})'

    def __str__(self):
        string = str(self.finite_expansion)
        if self.periodic_expansion:
            string = string[: -1] + '; '\
                     + str(self.periodic_expansion)[1 : -1] + '...]'
        return string
```

```python
ContinuedFraction()
ContinuedFraction([])
ContinuedFraction([0, 1])

print(ContinuedFraction([1, 2, 1]))

ContinuedFraction([0, 1], [1])
ContinuedFraction([0], [1, 1])
ContinuedFraction([0, 1], [1, 1])
ContinuedFraction([], [1, 1, 1])
ContinuedFraction([0, 1], [1, 1, 1])

print(ContinuedFraction([0], [1, 2, 1, 2, 1, 2]))
print(ContinuedFraction([0, 2], [1, 2, 1, 2, 1, 2]))
print(ContinuedFraction([0, 1, 2, 3], [4, 2, 3, 4, 2, 3]))
print(ContinuedFraction([0, 1, 2, 3, 1], [4, 2, 3, 1]))
```

[3]: ContinuedFraction([0], [])

[3]: ContinuedFraction([0], [])

[3]: ContinuedFraction([1], [])

```
[1, 3]
```

[3]: ContinuedFraction([0], [1])

[3]: ContinuedFraction([0], [1])

[3]: ContinuedFraction([0], [1])

[3]: ContinuedFraction([0], [1])

[3]: ContinuedFraction([0], [1])

```
[0; 1, 2…]
[0; 2, 1…]
[0, 1; 2, 3, 4…]
[0, 1; 2, 3, 1, 4…]
```

Checking whether a continued fraction represents an integer, or whether it represents a rational number, is straightforward. Given a continued fraction F, we compute the negation $\overline{F}$ of F as follows.

- If F is of the form $[a]$ then $\overline{F}$ is $[-a]$.
- If F is of the form $[a, b\ldots]$ with $b > 1$ then $\overline{F}$ is $[-a-1, 1, b-1\ldots]$.
- If F is of the form $[a, 1\ldots]$ then it is actually of the form $[a, 1, c\ldots]$, and then $\overline{F}$ is $[-a-1, 1+c\ldots]$.

If `L` is a `ContinuedFraction` object and `L.periodic_expansion` is not the empty list, then we can extend `L.finite_expansion` with one or two copies of `L.periodic_expansion` depending on whether the latter has a length greater than 1 or a length equal to 1, respectively. Then `L` is no longer normalised but represents the same continued fraction. This guarantees that:

- either `L.finite_expansion` has a length of 1 while `L.periodic_expansion` is empty
- or `L.finite_expansion` has a length of 2 and does not end in 1 while `L.periodic_expansion` is empty,
- or `L.finite_expansion` has a length of 3 at least.

This allows one to easily perform the three cases of the computation:

[4]:
```
class ContinuedFraction(ContinuedFraction):
    def is_integral(self):
        return len(self.finite_expansion) == 1\
               and not self.periodic_expansion

    def is_rational(self):
        return not self.periodic_expansion

    def negation(self):
        # In case the periodic expansion is not empty,  borrow from it
        # so as to make the length of the finite expansion at least 3,
        # as that simplifies the computation.
        if len(self.periodic_expansion) == 1:
            finite_expansion =\
                    self.finite_expansion + self.periodic_expansion * 2
        elif self.periodic_expansion:
            finite_expansion =\
                    self.finite_expansion + self.periodic_expansion
        else:
            finite_expansion = self.finite_expansion
        periodic_expansion = self.periodic_expansion
        if len(finite_expansion) == 1:
            return ContinuedFraction([-finite_expansion[0]])
        # In this case, finite_expansion is of length at least 3.
        if finite_expansion[1] == 1:
            return ContinuedFraction([-finite_expansion[0] - 1,
                                       1 + finite_expansion[2]
                                      ] + finite_expansion[3 :],
                                      periodic_expansion
                                     )
        return ContinuedFraction([-finite_expansion[0] - 1, 1,
                                  finite_expansion[1] - 1
                                 ] + finite_expansion[2 :], periodic_expansion
                                )


cf = ContinuedFraction([])
```

```
cf.is_integral(), cf.is_rational()

cf = ContinuedFraction([1])
cf.is_integral(), cf.is_rational()

cf = ContinuedFraction([1, 2])
cf.is_integral(), cf.is_rational()

cf = ContinuedFraction([1], [2])
cf.is_integral(), cf.is_rational()

ContinuedFraction().negation()
ContinuedFraction([1]).negation(), ContinuedFraction([-1]).negation()
ContinuedFraction([1, 3]).negation(), ContinuedFraction([-2, 1, 2]).negation()
ContinuedFraction([1, 2]).negation(), ContinuedFraction([-2, 2]).negation()
ContinuedFraction([1], [1]).negation(),\
    ContinuedFraction([-2, 2], [1]).negation()
ContinuedFraction([1, 3, 4], [5, 6]).negation(),\
    ContinuedFraction([-2, 1, 2, 4], [5, 6]).negation()
ContinuedFraction([0, 1], [2, 3]).negation(),\
    ContinuedFraction([-1, 3], [3, 2]).negation()
```

[4]: `(True, True)`

[4]: `(True, True)`

[4]: `(False, True)`

[4]: `(False, False)`

[4]: `ContinuedFraction([0], [])`

[4]: `(ContinuedFraction([-1], []), ContinuedFraction([1], []))`

[4]: `(ContinuedFraction([-2, 1, 2], []), ContinuedFraction([1, 3], []))`

[4]: `(ContinuedFraction([-2, 2], []), ContinuedFraction([1, 2], []))`

[4]: `(ContinuedFraction([-2, 2], [1]), ContinuedFraction([1], [1]))`

[4]:
```
(ContinuedFraction([-2, 1, 2, 4], [5, 6]),
 ContinuedFraction([1, 3, 4], [5, 6]))
```

[4]: `(ContinuedFraction([-1, 3], [3, 2]), ContinuedFraction([0, 1], [2, 3]))`

Evaluating a finite continued fraction $[a_0, \ldots, a_k]$ as a rational number, in the form of a fraction, is immediate based on (1) and the definition of the sequences $(p_i)_{-2 \leq i \leq k}$ and $(q_i)_{-2 \leq i \leq k}$ that precedes (1):

```
[5]: class ContinuedFraction(ContinuedFraction):
         def to_fraction(self):
             if not self.is_rational():
                 return
             p1, p2 = 0, 1
             q1, q2 = 1, 0
             for a in self.finite_expansion:
                 p1, p2 = p2, a * p2 + p1
                 q1, q2 = q2, a * q2 + q1
             return Fraction(p2, q2)

     ContinuedFraction().to_fraction()
     ContinuedFraction([0, 1]).to_fraction()
     ContinuedFraction([0, 2]).to_fraction()
     ContinuedFraction([0, 1, 1]).to_fraction()
     ContinuedFraction([2, 1, 4, 3]).to_fraction()
     ContinuedFraction([2, 1, 4, 2, 1]).to_fraction()
```

```
[5]: Fraction(0, 1)
```

```
[5]: Fraction(1, 1)
```

```
[5]: Fraction(1, 2)
```

```
[5]: Fraction(1, 2)
```

```
[5]: Fraction(45, 16)
```

```
[5]: Fraction(45, 16)
```

For infinite continued fractions $[a_0, a_1, a_2, a_3 ...]$, one can generate $[a_0]$, $[a_0, a_1]$, $[a_0, a_1, a_2]$, $[a_0, a_1, a_2, a_3]$... as fractions that provide better and better approximations to $[a_0, a_1, a_2, a_3 ...]$. The `cycle` and `chain` classes from the `itertools` module are all we need to, given a `ContinuedFraction` object `L`, generate on demand all elements in `L.finite_expansion`, and then all elements in `L.periodic_expansion` again and again. Indeed, `cycle` allows one to create an iterator from a finite sequence S to generate on demand the elements in S again and again, getting back to S's first element after S's last element has been generated, while `chain` allows one to create an iterator to generate on demand the elements in one sequence and then the elements in another sequence:

```
[6]: C_1 = cycle([2, 3, 4])
     list(next(C_1) for _ in range(10))

     C_2 = chain([0, 1], cycle([2, 3, 4]))
     list(next(C_2) for _ in range(10))
```

```
[6]: [2, 3, 4, 2, 3, 4, 2, 3, 4, 2]
```

```
[6]: [0, 1, 2, 3, 4, 2, 3, 4, 2, 3]
```

The method `approximate_as_fractions()` is then a variation on the method `to_fraction()` previously implemented, dealing with infinite continued fractions rather than finite ones:

[7]:
```
class ContinuedFraction(ContinuedFraction):
    def approximate_as_fractions(self):
        p1, p2 = 0, 1
        q1, q2 = 1, 0
        for a in chain(self.finite_expansion, cycle(self.periodic_expansion)):
            p1, p2 = p2, a * p2 + p1
            q1, q2 = q2, a * q2 + q1
            yield Fraction(p2, q2)

# sqrt(2)
fractions = ContinuedFraction([1], [2]).approximate_as_fractions()
for _ in range(10):
    print(next(fractions))

print()

# -sqrt(3)
fractions = ContinuedFraction([-2, 3], [1, 2]).approximate_as_fractions()
for _ in range(10):
    print(next(fractions))
```

```
1
3/2
7/5
17/12
41/29
99/70
239/169
577/408
1393/985
3363/2378

-2
-5/3
-7/4
-19/11
-26/15
-71/41
-97/56
-265/153
-362/209
-989/571
```

Let us extend the `Fraction` class with a method, `to_continued_fraction()`, that computes the (finite) continued fraction's representation of its argument as a `ContinuedFraction` object. For positive fractions, the implementation is straightforward, exploiting Euclid's algorithm as described at the beginning. For a negative fraction F, it suffices to apply `ContinuedFraction`'s `negation()` method to the continued fraction object computed from $-F$:

[8]:
```
class Fraction(Fraction):
    def to_continued_fraction(self):
        factors = []
        a, b = abs(self.numerator), self.denominator
        while b:
            factors.append(a // b)
            a, b = b, a % b
        if self.numerator >= 0:
            return ContinuedFraction(factors)
        return ContinuedFraction(factors).negation()

Fraction().to_continued_fraction()
Fraction(-2).to_continued_fraction()
Fraction(1, 2).to_continued_fraction()
Fraction(-8, 5).to_continued_fraction()
Fraction(15, 11).to_continued_fraction()
Fraction(-1080, 384).to_continued_fraction()
```

[8]:
```
ContinuedFraction([0], [])
```

[8]:
```
ContinuedFraction([-2], [])
```

[8]:
```
ContinuedFraction([0, 2], [])
```

[8]:
```
ContinuedFraction([-2, 2, 2], [])
```

[8]:
```
ContinuedFraction([1, 2, 1, 3], [])
```

[8]:
```
ContinuedFraction([-3, 5, 3], [])
```

Let us extend the `Fraction` class with a method, `approximate_as_decimals()`, that given a fraction F and a strictly positive integer ϖ, the precision, set by default to 1, yields strings s_1, s_2, s_3,… for the decimal representation of F with the following properties:

- if F is an integer then s_1, s_2, s_3,… represent that integer;
- for all $i \in \mathbf{N} \setminus \{0\}$, if F is not an integer and has fewer than $\varpi \times i$ digits after the decimal point, then s_i represents F perfectly;
- for all $i \in \mathbf{N} \setminus \{0\}$, if F is not an integer and has at least $\varpi \times i$ digits after the decimal point, then s_i represents F with exactly $\varpi \times i$ digits after the decimal point, which are all correct.

We intend `approximate_as_decimals()` to be a generator function. In case F is an integer, generating F again and again is conveniently achieved with the `repeat()` function from the `itertools` module:

[9]:
```
list(repeat(3, times=4))

print()
```

```
for _ in range(4):
    next(repeat(3))
```

[9]: [3, 3, 3, 3]

[9]: 3

[9]: 3

[9]: 3

[9]: 3

In case F is not an integer, one can compute F's decimals as done manually:

[10]:
```
p = 3
q = 130
x = p / q
print(x)

p = p % q * 10
p // q

p = p % q * 10
p // q

p = p % q * 10
p // q

p = p % q * 10
p // q

p = p % q * 10
p // q
```

0.023076923076923078

[10]: 0

[10]: 2

[10]: 3

[10]: 0

[10]: 7

If the decimal representation of F is finite then p eventually becomes equal to 0, at which point no new decimal digit is to be generated. The following generator function takes a more general approach, generating decimals in chunks of a given size, set to 1 by default, forever or until all decimal digits have been exhausted, then ending in a chunk of a smaller size that can possibly be empty:

[11]:
```
def precision_many_decimals(p, q, precision=1):
    while True:
        decimals = []
        for _ in range(precision):
            if not p:
                yield decimals
                return
            decimals.append(p // q)
            p = p % q * 10
        yield decimals

p = 1
q = 64
x = p / q
print(x)

for decimals in precision_many_decimals(p % q * 10, q):
    decimals

print()

for decimals in precision_many_decimals(p % q * 10, q, 2):
    decimals

print()

for decimals in precision_many_decimals(p % q * 10, q, 4):
    decimals
```

```
0.015625
```

[11]: [0]

[11]: [1]

[11]: [5]

[11]: [6]

[11]: [2]

[11]: [5]

[11]: []

[11]: [0, 1]

[11]: [5, 6]

[11]: [2, 5]

[11]: []

[11]: [0, 1, 5, 6]

[11]: [2, 5]

Putting things together, we implement the method `approximate_as_decimals()` of the extended `Fraction` class as follows. The `repeat()` generator function is used in two circumstances, namely, in case F is finite, and in case F is not finite but its decimal representation is finite:

[12]:
```
class Fraction(Fraction):
    def approximate_as_decimals(self, precision=1):
        if self.denominator == 1:
            yield from repeat(str(self.numerator // self.denominator))
        if self.numerator > 0:
            representation = str(self.numerator // self.denominator) + '.'
        else:
            representation =\
                    '-' + str(abs(self.numerator) // self.denominator) + '.'
        for decimals in self.precision_many_decimals(
                            abs(self.numerator) % self.denominator * 10,
                            self.denominator, precision
                                                    ):
            representation += ''.join(str(d) for d in decimals)
            yield representation
        yield from repeat(representation)

    def precision_many_decimals(self, p, q, precision):
        while True:
            decimals = []
            for _ in range(precision):
                if not p:
                    yield decimals
                    return
                decimals.append(p // q)
                p = p % q * 10
            yield decimals
```

```
decimals = Fraction().approximate_as_decimals()
[next(decimals) for _ in range(3)]

decimals = Fraction(-200).approximate_as_decimals(2)
[next(decimals) for _ in range(3)]

decimals = Fraction(1, 2).approximate_as_decimals()
[next(decimals) for _ in range(3)]

decimals = Fraction(1, 3).approximate_as_decimals(4)
[next(decimals) for _ in range(3)]

decimals = Fraction(-14, 30000).approximate_as_decimals(4)
[next(decimals) for _ in range(3)]

decimals = Fraction(3, 130).approximate_as_decimals(4)
[next(decimals) for _ in range(3)]
```

```
[12]: ['0', '0', '0']

[12]: ['-200', '-200', '-200']

[12]: ['0.5', '0.5', '0.5']

[12]: ['0.3333', '0.33333333', '0.333333333333']

[12]: ['-0.0004', '-0.00046666', '-0.000466666666']

[12]: ['0.0230', '0.02307692', '0.023076923076']
```

Finally, let us extend the `ContinuedFraction` class with a method, `approximate_as_decimals()`, that performs identically to the method `approximate_as_decimals()` of the `Fraction` class, but operating on objects of type `ContinuedFraction`. So we want that method to be a generator function that given a continued fraction r and a strictly positive integer ϖ, the precision, set by default to 1, yields strings s_1, s_2, s_3,... for the decimal representation of r with the following properties:

- if r is an integer then s_1, s_2, s_3,... represent that integer;
- for all $i \in \mathbf{N} \setminus \{0\}$, if r is not an integer and has fewer than $\varpi \times i$ digits after the decimal point, then s_i represents r perfectly;
- for all $i \in \mathbf{N} \setminus \{0\}$, if r is not an integer and has at least $\varpi \times i$ digits after the decimal point, then s_i represents r with exactly $\varpi \times i$ digits after the decimal point, which are all correct.

Let `cf` denote an object of type `ContinuedFraction` that represents r. In case r is rational then it suffices to call `to_fraction()` on `cf` to get a `Fraction` object and then call `approximate_as_decimals()` on the latter. Now suppose that r is not rational, hence the decimal representation of r is infinite. Its integral part is `cf.finite_expansion[0]` if r is positive and `cf.finite_expansion[0] + 1` otherwise (see Section 4). Let `fractions` denote `approximate_as_fractions()`, which returns a sequence of fractions $(F_0, F_1, F_2 \ldots)$. It follows from (4) and the observations that follow that the members of this sequence alternate between an approximation of r from below and an approximation of r from above, and that every second member offers a closer and closer approximation to r: one of $(F_0, F_2, F_4 \ldots)$ and $(F_1, F_3, F_5 \ldots)$ offers closer and closer approximations of r from below, while the other offers closer and closer approximations of r from above. Hence for all $i, p \in \mathbf{N} \setminus \{0\}$, if F_i and F_{i+1} agree on the first p decimal digits after the comma of their decimal expansions, then those p digits are the first p decimal digits after the comma of r's decimal expansion. (Note that F_0 is an integer, F_1 might be an integer (then necessarily equal to $F_0 + 1$), and F_2, F_3... are not integers. This is why it is necessary to ignore F_0 and start with $i = 1$.) So given $p \in \mathbf{N} \setminus \{0\}$, in order to compute the first p digits after the comma of the decimal expansion of r, it suffices to find $i \in \mathbf{N} \setminus \{0\}$ such that F_i and F_{i+1} agree on the first p decimal digits after the comma of their decimal expansions (which will then be the first p decimal digits after the comma of their decimal expansion of F_j for all $j > i$). These observations lead to the following adaptation of `Fraction`'s `approximate_as_decimals()` method of `ContinuedFraction`'s `approximate_as_decimals()` method:

[13]:
```python
class ContinuedFraction(ContinuedFraction):
    def approximate_as_decimals(self, precision=1):
        if self.is_rational():
            yield from self.to_fraction().approximate_as_decimals(precision)
        else:
            if self.finite_expansion[0] >= 0 or self.is_integral():
                representation = str(self.finite_expansion[0]) + '.'
            else:
                representation = str(self.finite_expansion[0] + 1) + '.'
            fractions = self.approximate_as_fractions()
            current_precision = precision
            # Ignore first fraction which is necessarily an integer.
            next(fractions)
            # Might be an integer, but next fraction will not be.
            fraction = next(fractions)
            s1 = next(fraction.precision_many_decimals(
                        abs(fraction.numerator) % fraction.denominator * 10,
                        fraction.denominator, current_precision
                                                      )
                     )
            while True:
                fraction = next(fractions)
                s2 = next(fraction.precision_many_decimals(
                        abs(fraction.numerator) % fraction.denominator * 10,
                        fraction.denominator, current_precision
                                                          )
                         )
                if s1 == s2:
                    representation += ''.join(str(d) for d in s1[-precision :])
                    yield representation
                    current_precision += precision
```

```
                s1 = s2

decimals = ContinuedFraction().approximate_as_decimals()
[next(decimals) for _ in range(3)]

decimals = ContinuedFraction([2]).approximate_as_decimals()
[next(decimals) for _ in range(3)]

decimals = ContinuedFraction([0, 2]).approximate_as_decimals()
[next(decimals) for _ in range(3)]

decimals = ContinuedFraction([0, 3]).approximate_as_decimals(4)
[next(decimals) for _ in range(3)]

decimals = ContinuedFraction([-1, 1, 2]).approximate_as_decimals(4)
[next(decimals) for _ in range(5)]

decimals = ContinuedFraction([0, 1000000], [1]).approximate_as_decimals(2)
[next(decimals) for _ in range(10)]

# Golden ratio
decimals = ContinuedFraction([1], [1]).approximate_as_decimals()
[next(decimals) for _ in range(10)]

# sqrt(2)
decimals = ContinuedFraction([1], [2]) .approximate_as_decimals(2)
[next(decimals) for _ in range(10)]

# -sqrt(3)
decimals = ContinuedFraction([-2, 3], [1, 2]).approximate_as_decimals(4)
[next(decimals) for _ in range(10)]
```

```
[13]: ['0', '0', '0']

[13]: ['2', '2', '2']

[13]: ['0.5', '0.5', '0.5']

[13]: ['0.3333', '0.33333333', '0.333333333333']

[13]: ['-0.3333',
       '-0.33333333',
       '-0.333333333333',
       '-0.3333333333333333',
       '-0.33333333333333333333']
```

```
[13]: ['0.00',
 '0.0000',
 '0.000000',
 '0.00000099',
 '0.0000009999',
 '0.000000999999',
 '0.00000099999938',
 '0.0000009999993819',
 '0.000000999999381966',
 '0.00000099999938196639']
```

```
[13]: ['1.6',
 '1.61',
 '1.618',
 '1.6180',
 '1.61803',
 '1.618033',
 '1.6180339',
 '1.61803398',
 '1.618033988',
 '1.6180339887']
```

```
[13]: ['1.41',
 '1.4142',
 '1.414213',
 '1.41421356',
 '1.4142135623',
 '1.414213562373',
 '1.41421356237309',
 '1.4142135623730950',
 '1.414213562373095048',
 '1.41421356237309504880']
```

```
[13]: ['-1.7320',
 '-1.73205080',
 '-1.732050807568',
 '-1.7320508075688772',
 '-1.73205080756887729352',
 '-1.732050807568877293527446',
 '-1.7320508075688772935274463415',
 '-1.73205080756887729352744634150587',
 '-1.732050807568877293527446341505872366',
 '-1.7320508075688772935274463415058723669428']
```