Contents

Problem 73. Counting reduced proper fractions

Given two rational numbers $0 < x < y < 1$ and a positive integer N, we want to find the number of reduced proper fractions $x < \frac{k}{n} < y$ with $n \leqslant N$.

73.1 The straightforward method

The direct approach is to check all fractions $\frac{k}{n}$ between x and y with $n \leqslant N$ whether they are reduced. If there are not too many pairs to test, the simple check and count loop, (with the problem's parameters $x = \frac{1}{3}$, $y = \frac{1}{2}$ and $N = 10\,000$ for concreteness),

```
limit := 10000
count := 0
for n := 5 to limit
    for k := n div 3 + 1 to (n - 1) div 2
        if gcd(n,k) = 1 then count := count + 1
    end for
end for
output count
```

where the greatest common divisor is calculated with the Euclidean algorithm

© Project Euler, further distribution without the consent of the author(s) prohibited
Author: daniel.is.fischer

```
function gcd(a, b)
    r := a mod b
    while not (r = 0)
        a := b
        b := r
        r := a mod b
    end while
    return b
end function
```

may suffice, for $N = 10\,000$ it will find the answer in a few seconds. But it scales badly. The number of fractions to check is proportional to N^2 and calculating the gcd also takes longer for larger inputs, the time complexity of this algorithm is $\mathcal{O}(N^2 \cdot \log N)$. It can be sped up by a significant factor with some easy optimisations. It is straightforward to avoid calling gcd with two even numbers, the case of two multiples of 3 can also be skipped by unrolling the loops. If the integer type supports direct bit-shifting, using the binary-GCD-algorithm instead of the Euclidean algorithm can be a substantial gain, but it remains a slow algorithm.

73.2 Farey Sequences

The Farey sequence (also called Farey series) of order m, $\mathcal{F}_m$, is the finite sequence of reduced fractions $0 \leqslant {}^{k}\!/_{n} \leqslant 1$ with denominator $n \leqslant m$, in ascending order. So, for example, $\mathcal{F}_1 = \left(\frac{0}{1}, \frac{1}{1}\right)$ and $\mathcal{F}_4 = \left(\frac{0}{1}, \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{1}{1}\right)$.

In other words, given $0 < x < y < 1$ and N, our task is to find the number of fractions in $\mathcal{F}_N$ between x and y.

We will present two algorithms of general interest based on the theory of Farey sequences. The key facts that we need are

(*i*) Two reduced fractions $0 \leqslant \frac{a}{b} < \frac{c}{d} \leqslant 1$ are neighbours in some Farey sequence if and only if $b \cdot c - a \cdot d = 1$.

(*ii*) If $b \cdot c - a \cdot d = 1$, the (unique) fraction between $\frac{a}{b}$ and $\frac{c}{d}$ with the smallest denominator is the *mediant* $\frac{a+c}{b+d}$ of these fractions.

(*iii*) If $\frac{a}{b} < \frac{c}{d} < \frac{e}{f}$ are three consecutive fractions in $\mathcal{F}_k$, then $\frac{c}{d} = \frac{a+e}{b+f}$.

The proofs are given in the appendix.

© Project Euler, further distribution without the consent of the author(s) prohibited
Author: daniel.is.fischer

73.2.1 Enumerating the fractions in order

The first algorithm enumerates the Farey sequence in (ascending) order. If $\frac{a}{b} < \frac{c}{d}$ are two consecutive Fractions in $\mathcal{F}_N$, facts (ii) and (iii) allow us to find the next larger fraction easily. Let that fraction be $\frac{e}{f}$.

By (iii), we know that there is a $k \geqslant 1$ with $a + e = k \cdot c$ and $b + f = k \cdot d$. By (ii), we know that $d + f > N$. Together, these imply $k \cdot d = b + f \leqslant b + N < b + f + d = (k + 1) \cdot d$, or

$$k = \left\lfloor \frac{N + b}{d} \right\rfloor, \tag{73.1}$$

thus $e = \left\lfloor \frac{N + b}{d} \right\rfloor \cdot c - a$ and $f = \left\lfloor \frac{N + b}{d} \right\rfloor \cdot d - b$.

To get started, we must set $\frac{a}{b}$ to x and $\frac{c}{d}$ to the right neighbour of x in $\mathcal{F}_N$. By facts (i) and (ii), $\frac{c}{d}$ is the right neighbour of $\frac{a}{b}$ in $\mathcal{F}_N$ if and only if $b \cdot c - a \cdot d = 1$ and $d \leqslant N < b + d$.

First, by the extended euclidean algorithm (or continued fraction expansion), we find $c_0 \leqslant a$ and $d_0 \leqslant b$ with $b \cdot c_0 - a \cdot d_0 = 1$. Since $\gcd(a, b) = 1$, we have $b \cdot c - a \cdot d = b \cdot c_0 - a \cdot d_0$ if and only if there is a $k \in \mathbb{Z}$ with $c = c_0 + k \cdot a$ and $d = d_0 + k \cdot b$. To have $d \leqslant N < b + d$, we must take $k = \left\lfloor \frac{N - d_0}{b} \right\rfloor$, giving

$$c = c_0 + \left\lfloor \frac{N - d_0}{b} \right\rfloor \cdot a, \qquad d = d_0 + \left\lfloor \frac{N - d_0}{b} \right\rfloor \cdot b. \tag{73.2}$$

For the given problem, we have $a = 1,\ b = 3$, hence $c_0 = 1,\ d_0 = 2$ and $k = 3332$, so $c = 3333,\ d = 9998$. The pseudocode for the algorithm is

© Project Euler, further distribution without the consent of the author(s) prohibited
Author: daniel.is.fischer

```
limit := 10000
a := 1
b := 3
c := 3333
d := 9998
count := 0
while not (c = 1 and d = 2)
    count := count + 1
    k := (limit + b) div d
    e := k*c - a
    f := k*d - b
    a := c
    b := d
    c := e
    d := f
end while
output count
```

The algorithm runs in constant small memory, its time complexity is $\mathcal{O}(N^2)$. It is much faster than the straightforward already for small N and scales a little better. Also, it can be useful for other purposes, so it's vastly superior.

If the fractions themselves aren't needed, a (very) small speedup can be obtained by ignoring the numerators.

73.2.2 The Stern-Brocot tree

The second algorithm enumerates the fractions in a different order, corresponding to a depth-first traversal of the Stern-Brocot tree, which is a frequently used strategy. The basis of it is fact (*ii*): the first fraction to go between two adjacent fractions in some Farey sequence is the mediant of the two, which here is reduced since $(a+c)\cdot b-(b+d)\cdot a = c\cdot b-d\cdot a = 1$.

So we can count the fractions between $l=\frac{a}{b}$ and $r=\frac{c}{d}$, where $c\cdot b-d\cdot a=1$ by inserting the mediant $m=\frac{e}{f}=\frac{a+c}{b+d}$ if $f\leqslant N$, counting the fractions between l and m in the same manner, then counting the fractions between m and r and adding the two subcounts plus one for the mediant; if $f=b+d>N$, there is no fraction between l and r in $\mathcal{F}_N$.

If the two limits aren't neighbours in any Farey sequence, i.e. if $c\cdot b-d\cdot a>1$, this method cannot directly be employed. One would then enumerate the fractions between l and r in $\mathcal{F}_{\max\{b,\,d\}}$ by the previous method and count the fractions between each adjacent pair in that sequence by the Stern-Brocot method.

© Project Euler, further distribution without the consent of the author(s) prohibited
Author: daniel.is.fischer

Of course, this mixed-method approach is only worthwhile if the Stern-Brocot method has manifest advantages over enumerating the fractions in ascending order, e.g. superior speed.

The counting function could then look like

```
function countSB(limit,leftN,leftD,rightN,rightD)
    medN := leftN + rightN
    medD := leftD + rightD
    if medD > limit
      then
        return 0
      else
        count := 1
        count := count + countSB(limit,leftN,leftD,medN,medD)
        count := count + countSB(limit,medN,medD,rightN,rightD)
        return count
end function
```

This algorithm also has time complexity $\mathcal{O}(N^2)$, it requires $\mathcal{O}(N)$ stack space, so in that respect the previous algorithm is better, but in practice the algorithm would need unbearably long time before the space requirements pose any problem. However, this algorithm uses only additions, no divisions, so if the function-call overhead is small enough, it will be faster than the previous. The relative speed of the two algorithms depends heavily on the used language, platform and compiler optimisation-levels. A factor of 3 in either direction should not be surprising.

As for the previous algorithm, only the denominators are really needed to count the fractions. Omitting the numerators gives a noticeable speedup here if the function-call overhead is small.

A larger speedup can be obtained by transforming the recursion into a loop, manually managing the stack. Knowing more about the algorithm than the compiler, we need to store less information and can replace a recursive call with one array write (two if we don't omit the numerators). Having eliminated the function-call costs, the omission of the numerators now has a bigger impact and reduces the running time by 20–25%.

Using all mentioned optimisations, the fastest code yet (at least in compiled languages) becomes

© Project Euler, further distribution without the consent of the author(s) prohibited

```
limit := 10000
count := 0
top := 0
stack := new int array [0 .. limit/2]
left := 3
right := 2
while true
    med := left + right
    if med > limit then
        if top > 0 then
            left := right
            top := top - 1
            right := stack[top]
        else
            break
        end if
    else
        count := count + 1
        stack[top] := right
        top := top + 1
        right := med
    end if
end while
output count
```

Among the algorithms which *count* each fraction, this is about as fast as you can get. It solves the problem for $N = 10\,000$ in a few dozen milliseconds and for $N = 100\,000$ in a few seconds. But the quadratic behaviour makes it unsuitable for large N.

73.3 Fast Algorithms

Since the number of reduced fractions between x and y is about $\frac{3N^2}{\pi^2} \cdot (y - x)$, a fast algorithm must determine the exact number by other means than counting.

There's a plethora of algorithms that avoid counting each fraction, many of them are interesting in some way, far too many to mention them all. We shall discuss only two of them, the most interesting and fastest[1].

[1] If you know a faster algorithm or an equally fast essentially different algorithm, I'd be very interested to learn about it.

© Project Euler, further distribution without the consent of the author(s) prohibited

For fixed $x < y$, let

$$\begin{aligned} F(N) &:= \operatorname{card}\Big\{\frac{k}{n}\colon x < \frac{k}{n} < y,\ n \leqslant N\Big\}, \\ R(N) &:= \operatorname{card}\Big\{\frac{k}{n}\colon x < \frac{k}{n} < y,\ n \leqslant N,\ \gcd(k,n) = 1\Big\}. \end{aligned} \tag{73.3}$$

The starting point for both algorithms is the identity

$$F(N) = \sum_{m=1}^{N} R\Big(\Big\lfloor\frac{N}{m}\Big\rfloor\Big). \tag{73.4}$$

To see this, note that the number of fractions $\dfrac{k}{n}$ with $n \leqslant N$ and $\gcd(k,n) = m$ is precisely $R\Big(\Big\lfloor\dfrac{N}{m}\Big\rfloor\Big)$.

Since $F(N)$ is easy to calculate, a clever rewriting of (73.4) can lead to an efficient way to calculate $R(N)$. In our case, we have

$$\begin{aligned} F(m) &= \sum_{n=1}^{m}\Big(\Big\lfloor\frac{n-1}{2}\Big\rfloor - \Big\lfloor\frac{n}{3}\Big\rfloor\Big) \\ &= q\cdot(3q-2+r) + \begin{cases} 1 & \text{if } r = 5 \\ 0 & \text{otherwise,} \end{cases} \end{aligned} \tag{73.5}$$

where $m = 6q + r$, $0 \leqslant r < 6$.
Similar formulae exist for every choice of limits x, y.

73.3.1 Möbius Inversion or the Inclusion-Exclusion Principle

By a generalisation of the *Möbius inversion formula* for *arithmetic functions*, we can rewrite (73.4) as

$$R(N) = \sum_{m=1}^{N} \mu(m)\cdot F\Big(\Big\lfloor\frac{N}{m}\Big\rfloor\Big), \tag{73.6}$$

where μ is the *Möbius function*, given by

$$\mu(n) = \begin{cases} (-1)^r & \text{if } n \text{ is the product of } r \text{ distinct primes} \\ 0 & \text{if } n \text{ is not squarefree.} \end{cases} \tag{73.7}$$

Since 1 is the product of 0 primes, $\mu(1) = (-1)^0 = 1$.

© Project Euler, further distribution without the consent of the author(s) prohibited
Author: daniel.is.fischer

Another way to look at it and obtain (73.6) is the inclusion-exclusion principle. For $m \geqslant 1$, let

$$S(N,m) := \Big\{\frac{k}{n}\colon x < \frac{k}{n} < y,\ n \leqslant N,\ m \text{ divides } \gcd(k,n)\Big\}.$$

Then $F\Big(\Big\lfloor\frac{N}{m}\Big\rfloor\Big)$ is the cardinality of $S(N,m)$ and for any two m_1, m_2, we have $S(N,m_1) \cap S(N,m_2) = S\big(N, \operatorname{lcm}(m_1,m_2)\big)$.

Since a fraction is *not* reduced if and only if some prime divides its numerator and denominator, we find

$$R(N) = F(N) - \operatorname{card}\Big(\bigcup_p S(N,p)\Big), \tag{73.8}$$

where the union ranges over all small enough primes (in our case, $p \leqslant N/5$). By the inclusion-exclusion principle and the facts mentioned after the definition of $S(N,m)$, this expands to (73.6).

If the values of the Möbius function are available, (73.6) translates directly to an $\mathcal{O}(N)$ loop, but otherwise it is better to calculate $R(N)$ by a direct application of the inclusion-exclusion principle.

Assuming the relevant primes precalculated and stored in an array, the core of the algorithm is the function

```
function inclusionExclusion(limit,index)
    count := F(limit)   // see (73.5) for F
    while index < primeCount and 5*primes[index] <= limit
        newLimit := limit div primes[index]
        count := count - inclusionExclusion(newLimit,index+1)
        index := index + 1
    end while
    return count
end function
```

$R(N)$ is then calculated by `inclusionExclusion(N,0)`, if the prime-array has 0-based indices.

If the prime generation has time complexity $\mathcal{O}(N)$ or better, this is an $\mathcal{O}(N)$ algorithm which can solve the problem for $N = 10^9$ in about 15 seconds. As it stands, it requires $\mathcal{O}(N/\log N)$ memory, by a small modification this can be reduced to $\mathcal{O}(\sqrt{N}/\log N)$ without significant influence on the speed – the code becomes more complicated since stored and unstored primes have to be treated differently.

© Project Euler, further distribution without the consent of the author(s) prohibited
Author: daniel.is.fischer

73.3.2 A sublinear algorithm

Again we begin by rewriting (73.4). The first step of the rewriting consists simply of moving some terms to the other side of the equation:

$$R(N) = F(N) - \sum_{m=2}^{N} R\Big(\Big\lfloor \frac{N}{m} \Big\rfloor\Big). \tag{73.9}$$

In the second step, we replace $R\Big(\Big\lfloor \frac{N}{2} \Big\rfloor\Big)$ by the corresponding expression to obtain

$$\begin{aligned} R(N) &= F(N) - F\Big(\Big\lfloor \frac{N}{2} \Big\rfloor\Big) + \sum_{m=2}^{\lfloor \frac{N}{2} \rfloor} R\Big(\Big\lfloor \frac{N}{2m} \Big\rfloor\Big) - \sum_{m=3}^{N} R\Big(\Big\lfloor \frac{N}{m} \Big\rfloor\Big) \\ &= F(N) - F\Big(\Big\lfloor \frac{N}{2} \Big\rfloor\Big) - \sum_{k=1}^{\lfloor \frac{N-1}{2} \rfloor} R\Big(\Big\lfloor \frac{N}{2k+1} \Big\rfloor\Big). \end{aligned} \tag{73.10}$$

If we continue to replace $R\Big(\Big\lfloor \frac{N}{m} \Big\rfloor\Big)$ with the expression corresponding to (73.9), we will arrive at (73.6), but stopping now allows us to find a more efficient algorithm.

The crucial facts are that for large N, the expression $\Big\lfloor \frac{N}{2k+1} \Big\rfloor$ is constant for a long range of k (for sufficiently large k) and that all $R\Big(\Big\lfloor \frac{N}{2k+1} \Big\rfloor\Big)$ have the same coefficient in (73.10). This allows us to group the constant ranges, which can't be done in (73.6) because the sum of $\mu(n)$ over a range cannot easily be determined.

Now let us see how many distinct $R(m)$ appear in (73.10) and how often each occurs. We begin with the largest k (which corresponds to the smallest m) and work our way down. The largest k to consider is $k = \Big\lfloor \frac{N-1}{2} \Big\rfloor$, so $2k+1$ is then either N or $N-1$. For $N > 2$, the corresponding $m = \Big\lfloor \frac{N}{2k+1} \Big\rfloor$ is then 1. We can only skip an m if

$$\frac{N}{2k-1} - \frac{N}{2k+1} > 1 \iff k \leqslant \sqrt{\frac{N}{2}},$$

so with $k_0 := \Big\lfloor \sqrt{\frac{N}{2}} \Big\rfloor$, all $m \leqslant m_0 := \Big\lfloor \frac{N}{2k_0+1} \Big\rfloor$ occur and the $\Big\lfloor \frac{N}{2k+1} \Big\rfloor$ for $k < k_0$ are all unique. m_0 is one of $k_0 - 1, k_0$ or $k_0 + 1$, hence there appear at

© Project Euler, further distribution without the consent of the author(s) prohibited
Author: daniel.is.fischer

most $2k_0 \leqslant \sqrt{2N}$ distinct $R(m)$ on the right hand side of (73.10) and they split into two groups. We now determine the largest k so that $m \leqslant \left\lfloor \frac{N}{2k+1} \right\rfloor$:

$$\begin{aligned} m \leqslant \left\lfloor \frac{N}{2k+1} \right\rfloor &\iff m \leqslant \frac{N}{2k+1} \\ \iff 2k+1 \leqslant \frac{N}{m} &\iff k \leqslant \left\lfloor \frac{N/m - 1}{2} \right\rfloor. \end{aligned} \tag{73.11}$$

Let $k_{\max}(m) := \left\lfloor \frac{N/m - 1}{2} \right\rfloor$. Then $R(m)$ occurs $k_{\max}(m) - k_{\max}(m+1)$ times and we can write (73.10) as

$$\begin{aligned} R(N) = F(N) - F\Big(\Big\lfloor \frac{N}{2} \Big\rfloor\Big) - \sum_{k=1}^{k_0-1} R\Big(\Big\lfloor \frac{N}{2k+1} \Big\rfloor\Big) \\ - \sum_{m=1}^{m_0} \big(k_{\max}(m) - k_{\max}(m+1)\big) \cdot R(m). \end{aligned} \tag{73.12}$$

To calculate $R(N)$, we first must calculate $R(n)$ for several $n < N$. For each of these n (except $n < 3$) we must have calculated $R(m)$ for several $m < n$ before. For $R(n)$ we need $R\Big(\Big\lfloor \frac{n}{2j+1} \Big\rfloor\Big)$, $1 \leqslant j \leqslant \frac{n-1}{2}$. But $n = \Big\lfloor \frac{N}{2k+1} \Big\rfloor$ for some k, hence

$$\Big\lfloor \frac{n}{2j+1} \Big\rfloor = \Big\lfloor \frac{\lfloor N/(2k+1) \rfloor}{2j+1} \Big\rfloor = \Big\lfloor \frac{N}{(2k+1)(2j+1)} \Big\rfloor,$$

thus to calculate $R(n)$ we need no values which we did not directly need to calculate $R(N)$, about $\sqrt{2N}$ values altogether. Since they are used many times in general, we store the calculated values, hence the algorithm requires $\mathcal{O}(\sqrt{N})$ memory. If we calculate the needed $R(m)$ in ascending order, each $R(m)$ contributes $\mathcal{O}(\sqrt{m})$ operations, giving an overall time complexity of $\mathcal{O}(N^{3/4})$ – if storing and retrieving the values are constant time operations.

How to make the lookup a constant time operation is not obvious. For small enough N, we could store $R(m)$ at index m in an array of size N. This would waste a lot of space and hit the resource limits rather soon. While N is not too large, it is simple and very fast. Storing the values in a binary tree doesn't waste space, but the lookup is not constant time, this would change the complexity to $\mathcal{O}(N^{3/4} \cdot \log N)$, which is still excellent, but noticeably slower for large N.

However, we saw above that we need the values $R(n)$ for

(*i*) $1 \leqslant n \leqslant m_0$ and

© Project Euler, further distribution without the consent of the author(s) prohibited
Author: daniel.is.fischer

(*ii*) $n = \left\lfloor \frac{N}{2k+1} \right\rfloor$, $0 \leqslant k < k_0$.

Thus if we store the value $R(n), n \leqslant m_0$, in one array at index n and the values $R\Big(\left\lfloor \frac{N}{2k+1} \right\rfloor\Big), 0 \leqslant k < k_0$, in another array at index k, the lookup is $\mathcal{O}(1)$.

In our case we have $R(m) = 0$ for $m < 5$, which saves a little work – hardly measurable, though. In pseudocode, the algorithm becomes

```
N := 10000       // or 1000000000
K := ⌊√(N/2)⌋
M := ⌊N/(2K+1)⌋
rsmall := new int array [0 .. M] 0
rlarge := new int array [0 .. K-1] 0
```

Some global variables we need to reference, then the function counting the number of fractions between x ($= 1/3$) and y ($= 1/2$) with denominator $\leqslant n$, cf. (73.5):

```
function F(n)
    q := n div 6
    r := n mod 6
    f := q*(3*q - 2 + r)
    if r = 5 then f := f + 1
    return f
end function
```

Next, we translate (73.12) into a procedure which calculates and stores the number of reduced fractions between x and y with denominator $\leqslant n$, provided that the needed values of $R(m)$ have previously been calculated and stored:

© Project Euler, further distribution without the consent of the author(s) prohibited
Author: daniel.is.fischer

```
procedure R(n)
    switch := ⌊√(n/2)⌋
    count := F(n)
    count := count - F(n div 2)
    m := 5
    k := (n - 5) div 10
    while k ⩾ switch
        nextk := (n div (m + 1) - 1) div 2
        count := count - (k - nextk)*rsmall[m]
        k := nextk
        m := m+1
    end while
    while k > 0
        m := n div (2*k+1)
        if m ⩽ M then
            count := count - rsmall[m]
        else
            count := count - rlarge[((N div m) - 1) div 2]
        end if
        k := k - 1
    end while
    if n ⩽ M then
        rsmall[n] := count
    else
        rlarge[((N div n) - 1) div 2] := count
    end if
end procedure
```

The main programme is then

```
for n := 5 to M
    R(n)
end for
for j := K - 1 downto 0
    R(N div (2*j + 1))
end for
count := rlarge[0]
output count
```

This algorithm can find $R\left(10^{10}\right) = 5066059182147707040$ in under three seconds and $R\left(10^{12}\right) = 50660591821176634005758$ in just under ninety seconds.

© Project Euler, further distribution without the consent of the author(s) prohibited
Author: daniel.is.fischer

73.4 Appendix: Proofs

73.4.1 Farey Sequences

For convenience, we repeat the facts we are about to prove:

Proposition 73.1 *(i) Two reduced fractions* $0 \leqslant \frac{a}{b} < \frac{c}{d} \leqslant 1$ *are neighbours in some Farey sequence if and only if* $b \cdot c - a \cdot d = 1$.

(ii) If $b \cdot c - a \cdot d = 1$, *the (unique) fraction between* $\frac{a}{b}$ *and* $\frac{c}{d}$ *with the smallest denominator is the* mediant $\frac{a+c}{b+d}$ *of these fractions.*

(iii) If $\frac{a}{b}$, $\frac{c}{d}$ *and* $\frac{e}{f}$ *are three consecutive fractions in* $\mathcal{F}_k$, *then* $\frac{c}{d} = \frac{a+e}{b+f}$.

We first prove *(ii)*, so let $b \cdot c - a \cdot d = 1$ and $\frac{a}{b} < \frac{r}{s} < \frac{c}{d}$. Now

$$\begin{aligned} \frac{1}{b \cdot d} = \frac{c}{d} - \frac{a}{b} &= \left(\frac{c}{d} - \frac{r}{s}\right) + \left(\frac{r}{s} - \frac{a}{b}\right) \\ &= \frac{s \cdot c - r \cdot d}{s \cdot d} + \frac{r \cdot b - s \cdot a}{s \cdot b} \\ &\geqslant \frac{1}{s \cdot d} + \frac{1}{s \cdot b} = \frac{b+d}{s \cdot b \cdot d}, \end{aligned} \tag{73.13}$$

therefore $s \geqslant b+d$ and $s = b+d$ if and only if $s \cdot c - r \cdot d = 1 = r \cdot b - s \cdot a$. Since

$$(b+d) \cdot c - r \cdot d = 1 = b \cdot c - a \cdot d \iff r = a + c,$$

there is only one fraction with denominator $b+d$ between $\frac{a}{b}$ and $\frac{c}{d}$.

Using *(ii)*, the first proposition follows easily. On the one hand, *(ii)* immediately implies that if $b \cdot c - a \cdot d = 1$, then $\frac{a}{b}$ and $\frac{c}{d}$ are neighbours in all Farey sequences $\mathcal{F}_k$, $\max\{b, d\} \leqslant k < b + d$. The converse is shown by induction. Obviously $1 \cdot 1 - 0 \cdot 1 = 1$, so the condition holds for the neighbours in $\mathcal{F}_1$. Now assume it holds for all neighbours in $\mathcal{F}_k$ and let $\frac{a}{b}$, $\frac{c}{d}$ be arbitrary neighbours in $\mathcal{F}_k$. Then either $k+1 < b+d$ and they are also neighbours in $\mathcal{F}_{k+1}$ or $k+1 = b+d$ and the mediant splits them into two pairs of neighbours in $\mathcal{F}_{k+1}$, $\left(\frac{a}{b}, \frac{a+c}{b+d}\right)$ and $\left(\frac{a+c}{b+d}, \frac{c}{d}\right)$, which also satisfy the condition. Since all pairs of neighbours in $\mathcal{F}_{k+1}$ have at least one member already in $\mathcal{F}_k$, we are done.

Finally, *(iii)* is proved by induction, too. For $\mathcal{F}_2$, we have $\frac{1}{2} = \frac{0+1}{1+1}$. Now consider three consecutive fractions in $\mathcal{F}_k$, $k \geqslant 3$. If all three are already in

© Project Euler, further distribution without the consent of the author(s) prohibited
Author: daniel.is.fischer

$\mathcal{F}_{k-1}$, (iii) holds by the inductive hypothesis. If the middle fraction is new, we even have $c = a + e$ and $d = b + f$ by (ii). If exactly one of the outer fractions is new, without loss of generality assume $f = k$, let $\frac{g}{h}$ be the neighbour to the right of $\frac{c}{d}$ in $\mathcal{F}_{k-1}$. By the inductive hypothesis, $\frac{c}{d} = \frac{a+g}{b+h}$. By (ii), $e = c + g$ and $f = d + h$, so

$$(73.14)\quad d{\cdot}(a{+}e){-}c{\cdot}(b{+}f) = d{\cdot}(a{+}g{+}c){-}c{\cdot}(b{+}h{+}d) = d{\cdot}(a{+}g){-}c{\cdot}(b{+}h) = 0.$$

If both outer fractions are new $\big($which happens if and only if k is odd and $\frac{c}{d} = \frac{1}{2}\big)$, we write each of them as the mediant of the middle fraction and its previous neighbour and obtain the result by the same calculation.

© Project Euler, further distribution without the consent of the author(s) prohibited
Author: daniel.is.fischer