Problem 72. Find the number of reduced proper fractions whose denominator does not exceed N

The simplest method to determine that number is counting, just check every proper fraction whose denominator is $\leqslant N$ if it is reduced, i.e. if numerator and denominator are coprime. But once again, the simplest method is rather inefficient. Calculating greatest common divisors is a time-consuming task, and although we don't need to know the full gcd but only whether it is 1 or not, which allows to save a little work, the difference is minute. As there are $\frac{N(N-1)}{2}$ proper fractions to examine, this approach would take several hours and not a few minutes for $N = 1,000,000$, even when using the fact that $\frac{k}{n}$ is reduced if and only if $\frac{n-k}{n}$ is reduced, so halving the work.

We can do it orders of magnitude faster if we group the fractions by denominator instead of examining them individually. The set of reduced proper fractions with denominator n is $\left\{\frac{k}{n} : 1 \leqslant k \leqslant n \wedge \gcd(n,k) = 1\right\}$, so their number is $\varphi(n)$, where φ is Euler's totient function. A good method to calculate $\varphi(n)$ therefore leads to a fast way to the solution

$$\sum_{n=2}^{N} \varphi(n) \tag{72.1}$$

of our problem.

So, how do we calculate $\varphi(n)$? If k is *not* coprime to n, there is a prime p dividing n as well as k. Let $p_1, \ldots, p_r$ be the distinct primes dividing n. For each p_ϱ there are $\frac{n}{p_\varrho}$ numbers $\leqslant n$ divisible by p_ϱ. If $r = 1$, i.e. n is a prime power, thus $\varphi(n) = n - \frac{n}{p_1} = n \cdot \left(1 - \frac{1}{p_1}\right)$. If $r = 2$, there are $\frac{n}{p_1}$ numbers divisible by p_1 and $\frac{n}{p_2}$ divisible by p_2. But the $\frac{n}{p_1 p_2}$ numbers divisible by both primes have been counted twice, so we must subtract that number, leaving $\frac{n}{p_1} + \frac{n}{p_2} - \frac{n}{p_1 p_2}$ numbers not coprime to n, therefore then $\varphi(n) = n - \frac{n}{p_1} - \frac{n}{p_2} + \frac{n}{p_1 p_2} = n \cdot \left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)$. If $r = 3$, in the sum $\frac{n}{p_1} + \frac{n}{p_2} + \frac{n}{p_3}$ the numbers divisible by two of the primes are counted twice and those divisible by all three are counted thrice. For each selection of two of the three primes, we must subtract the count of the numbers divisible by both, giving $\frac{n}{p_1} + \frac{n}{p_2} + \frac{n}{p_3} - \frac{n}{p_1 p_2} - \frac{n}{p_1 p_3} - \frac{n}{p_2 p_3}$. But now we've subtracted the numbers divisible by all three primes thrice, hence we must again

Author: daniel.is.fischer

add that count, leading to $\varphi(n) = n - \frac{n}{p_1} - \frac{n}{p_2} - \frac{n}{p_3} + \frac{n}{p_1p_2} + \frac{n}{p_1p_2} + \frac{n}{p_1p_2} - \frac{n}{p_1p_2p_3}$. In general, by the inclusion-exclusion principle, we obtain the formula

$$\varphi(n) = n \cdot \prod_{p \,\big|\, n} \Big(1 - \frac{1}{p}\Big), \tag{72.2}$$

where $a \,\big|\, b$ means that a divides b.

Thus with

```
function eulerPhi(n)
    phi := n
    i := 1
    p := primes[1]
    while (i ⩽  primeCount and p*p  ⩽ n)
        if n mod p = 0 then
            phi := phi - phi div p
            while n mod p = 0
                n := n div p
            end while
        end if
        i := i+1
        p := primes[i]
    end while
    if n > 1 then
        phi := phi - phi div n
    end if
    return phi
end function
```

where `primes` is a list of the `primeCount` primes up to $\sqrt{N}$, we can calculate

```
fractionCount := 0
for n := 2 to N
    fractionCount := fractionCount + eulerPhi(n)
end for
output fractionCount
```

not too slowly. But this is still far from optimal. Division is a relatively slow operation and in this case, factorisation by trial division is especially wasteful, because we want to factorise a large block of consecutive numbers. This situation calls for a sieving approach. Formula (72.2) suggests a fairly simple approach:

(*i*) start with an array where the k^{th} entry is k,

(*ii*) for each prime p multiply the entries at multiples of p with $\left(1-\frac{1}{p}\right)$.

Primes are easy to detect with this approach, they are those numbers where the array entry has not been modified when we reach them, thus we arrive at the pseudocode

```
limit := 1000000
phis := new int array [2 .. limit]
for n := 2 to limit          // initialise array
    phis[n] := n
end for
for n := 2 to limit
    if phis[n] = n then      // n is a prime, for all multiples
                             // of n multiply with (1-1/n)
        for m := n to limit with step n
            phis[m] := phis[m] - phis[m] div n
        end for
    end if
end for
fractionCount := 0
for n := 2 to limit
    fractionCount := fractionCount + phis[n]    // add φ(n)
end for
output fractionCount
```

This algorithm is relatively simple and already rather fast, but we can get a significant speedup without becoming much more complicated.

With the prime factorisation $n = \prod {p_i}^{\alpha_i}$ we can rewrite (72.2) as

$$\varphi(n) = \prod (p_i - 1) \cdot {p_i}^{\alpha_i - 1}. \tag{72.3}$$

Now consider n and let p be a prime factor of n, $m = \frac{n}{p}$. Then by (72.3) you see

$$\varphi(n) = \varphi(m) \cdot \begin{cases} p & \text{, if } p \text{ divides } m, \\ (p-1) & \text{, otherwise.} \end{cases} \tag{72.4}$$

Thus we can quickly calculate the totients by first sieving the smallest prime factors and looking if that is a multiple factor of n:

Author: daniel.is.fischer

```
limit := 1000000
sieveLimit := ⌊√limit⌋
spf := new int array [2 .. limit]
for n := 2 to limit
    if n mod 2 = 0 then spf[n] := 2 else spf[n] := n
end for
for n := 3 to sieveLimit with step 2
    if spf[n] = n then      // n is a prime
        for m := n*n to limit with step 2*n
            if spf[m] = m then
                // n is the smallest prime factor of m
                spf[m] := n
            end if
        end for
    end if
end for
phis := new int array [2 .. limit]
for n := 2 to limit
    if spf[n] = n then      // n is a prime, so φ(n) = n − 1
        phis[n] := n-1
    else
        p := spf[n]
        m := n div p
        if spf[m] = p then  factor := p  else  factor := p-1
        phis[n] := factor*phis[m]
    end if
end for
fractionCount := 0
for n := 2 to limit
    fractionCount := fractionCount + phis[n]
end for
output fractionCount
```

The last algorithm, as it stands, needs twice the memory of the other sieve, which could be problematic for large N. This can be avoided, at the expense of slightly less legible code, by using the same array for the smallest prime factors and the totients, the `factor` would then be determined by checking if `m mod p = 0`, apart from that only the array names need to be adapted.

Both sieving methods can be optimised further, by sieving only odd numbers the memory requirements are halved and the execution time is also significantly reduced. If only odd numbers are sieved, the first thing that comes to mind for treating the even numbers is to treat them in the order they appear, dividing out

all factors 2 and using

$$\varphi\big(2^k \cdot n\big) = 2^{k-1} \cdot \varphi(n) \tag{72.5}$$

for $k > 0$ and odd n. But it's better to treat all even numbers with the same odd cofactor at once. Since

$$\sum_{k=0}^{m} \varphi\big(2^k \cdot n\big) = \left(1 + \sum_{k=1}^{m} 2^{k-1}\right)\varphi(n) = 2^m \varphi(n) \tag{72.6}$$

for any odd n, we can add all these totients at the same time, using just one multiplication by a power of 2 (a very fast bitshift, in many languages) and one addition instead of several additions and divisions (by powers of 2, those could be bitshifts, too), once we know the largest m with $2^m n \leqslant N$.
Lastly, since the totients are generated in order, there's no need for the separate summation pass, we can add them on the fly. With all these optimisations, the code becomes a little less than obvious.

```
limit := 1000000
sieveLimit := (⌊√limit⌋ - 1) div 2
maxIndex := (limit - 1) div 2
cache := new int array [1 .. maxIndex]
for n := 1 to maxIndex cache[n] := 0
for n := 1 to sieveLimit
    if cache[n] = 0 then    // 2*n + 1 is prime
        p := 2*n + 1
        for k := 2*n*(n+1) to maxIndex with step p
            if cache[k] = 0 then cache[k] = p
        end for
    end if
end for
multiplier := 1
while multiplier ⩽ limit
    multiplier := multiplier*2
end while
multiplier := multiplier div 2
fractionCount := multiplier - 1    // powers of 2
multiplier := multiplier div 2
stepIndex := ((limit div multiplier) + 1) div 2
```

In the initialisation part, we first sieve the smallest prime factors, leaving the n^{th} array entry zero if $2n + 1$ is prime. Then we determine the largest power of 2 not exceeding N – `multiplier` $= 2^m$ – and add the number of reduced proper

fractions whose denominator is a power of 2, by (72.6) and since we mustn't count $\frac{1}{1}$, that number is $2^m - 1$. Then `multiplier` is set to 2^{m-1} and `stepIndex` to the smallest index `n` so that $(2n+1) \cdot 2^{m-1} > N$. In the calculation loop, when we reach the step index, we set multiplier to the next smaller power of 2 and calculate the next step index, maintaining the invariant

$$(2n+1) \cdot \texttt{multiplier} \leqslant N < (2n+1) \cdot 2 \cdot \texttt{multiplier}.$$

Then $\varphi(2n+1)$ is calculated and the sum of the totients of all $2^k(2n+1)$ not exceeding N is added.

```
for n := 1 to maxIndex
    if n = stepIndex then
        multiplier := multiplier div 2
        stepIndex := ((limit div multiplier) + 1) div 2
    end if
    if cache[n] = 0 then
        cache[n] := 2*n
        fractionCount := fractionCount + multiplier*cache[n]
    else
        p := cache[n]
        cofactor := (2*n + 1) div p
        if cofactor mod p = 0 then factor := p else factor := p-1
        cache[n] := factor*cache[cofactor div 2]
        fractionCount := fractionCount + multiplier*cache[n]
    end if
end for
output fractionCount
```

There is a method to solve this problem without calculating the totients, which is much faster and uses less memory than the sieves presented here. That method will be discussed in the overview for problem 73.

Author: daniel.is.fischer