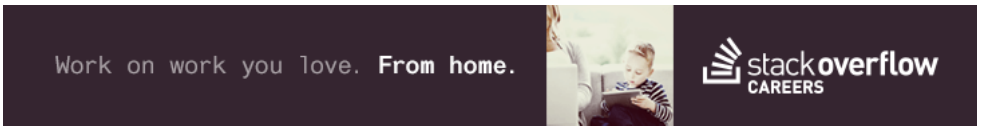
How many numbers below N are coprimes to N?



In short:

Given that **a** is coprime to **b** if **GCD(a,b) = 1** (where GCD stands for great common divisor), how many positive integers below N are coprime to N?

Is there a clever way?

Not necessary stuff

Here is the dumbest way:

```
def count_coprime(N):
    counter = 0
    for n in xrange(1,N):
        if gcd(n,N) == 1:
            counter += 1
    return counter
```

It works, but it is slow, and dumb. I'd like to use a clever and faster algorithm. I tried to use prime factors and divisors of N but I always get something that doesn't work with larger N.

~~I think the algorithm should be able to count them without calculating all of them like the dumbest algorithm does :P~~

Edit

It seems I've found a working one:

```
def a_bit_more_clever_counter(N):
    result = N - 1
    factors = []
    for factor, multiplicity in factorGenerator(N):
        result -= N/factor - 1
        for pf in factors:
            if lcm(pf, factor) < N:
                result += N/lcm(pf, factor) - 1
        factors += [factor]
    return result
```

where lcm is least common multiple. Does anyone have a better one?

Note

I'm using python, I think code should be readable even to who doesn't know python, if you find anything that is not clear just ask in the comments. I'm interested in the algorithm and the math, the idea.

algorithm math

edited Jun 19 '09 at 17:15

asked Jun 19 '09 at 17:09
Andrea Ambu
9,426 10 38 66

7 Homework? Project Euler? – starblue Jun 19 '09 at 17:54

4 Note that as some of the answers below have pointed out, this is Euler's totient. If you can find a truly clever way, NSA would be more than happy to know (as if you can efficiently find phi(n), you can solve the RSA public key algorithm) – Foon Jan 21 '13 at 15:36

4 Answers

[Edit] One last thought, which (IMO) is important enough that I'll put it at the beginning: if you're collecting a bunch of totients at once, you can avoid a lot of redundant work. Don't bother starting from large numbers to find their smaller factors -- instead, iterate over the smaller factors and accumulate results for the larger numbers.

```
class Totient:
    def __init__(self, n):
        self.totients = [1 for i in range(n)]
        for i in range(2, n):
            if self.totients[i] == 1:
                for j in range(i, n, i):
                    self.totients[j] *= i - 1
                    k = j / i
                    while k % i == 0:
                        self.totients[j] *= i
                        k /= i
    def __call__(self, i):
        return self.totients[i]
if __name__ == '__main__':
    from itertools import imap
    totient = Totient(10000)
    print sum(imap(totient, range(10000)))
```

This takes just 8ms on my desktop.

The Wikipedia page on the Euler totient function has some nice mathematical results.

$\sum_{d|n}\varphi(d)$ counts the numbers coprime to and smaller than each divisor of n: this has a trivial* mapping to counting the integers from 1 to n, so the sum total is n.

** by the second definition of* trivial

This is perfect for an application of the Möbius inversion formula, a clever trick for inverting sums of this exact form.

$$\varphi(n)=\sum_{d|n}d\cdot\mu\left(\frac{n}{d}\right)$$

This leads naturally to the code

```
def totient(n):
    if n == 1: return 1
    return sum(d * mobius(n / d) for d in range(1, n+1) if n % d == 0)
def mobius(n):
    result, i = 1, 2
    while n >= i:
        if n % i == 0:
            n = n / i
            if n % i == 0:
                return 0
            result = -result
        i = i + 1
    return result
```

There exist better implementations of the Möbius function, and it could be memoized for speed, but this should be easy enough to follow.

The more obvious computation of the totient function is

$$\varphi\left(p_1^{k_1}\cdots p_r^{k_r}\right)=(p_1-1)p_1^{k_1-1}\cdots(p_r-1)p_r^{k_r-1}=p_1^{k_1}\cdots p_r^{k_r}\prod_{i=1}^{r}\left(1-\frac{1}{p_r}\right)$$

In other words, fully factor the number into unique primes and exponents, and do a simple multiplication from there.

```
from operator import mul
def totient(n):
    return int(reduce(mul, (1 - 1.0 / p for p in prime_factors(n)), n))
def primes_factors(n):
    i = 2
    while n >= i:
        if n % i == 0:
            yield i
            n = n / i
            while n % i == 0:
                n = n / i
        i = i + 1
```

Again, there exist better implementations of `prime_factors`, but this is meant for easy reading.

```
# helper functions

from collections import defaultdict
from itertools import count
from operator import mul
def gcd(a, b):
    while a != 0: a, b = b % a, a
    return b
def lcm(a, b): return a * b / gcd(a, b)
primes_cache, prime_jumps = [], defaultdict(list)
def primes():
    prime = 1
    for i in count():
        if i < len(primes_cache): prime = primes_cache[i]
        else:
            prime += 1
            while prime in prime_jumps:
                for skip in prime_jumps[prime]:
                    prime_jumps[prime + skip] += [skip]
                del prime_jumps[prime]
                prime += 1
            prime_jumps[prime + prime] += [prime]
            primes_cache.append(prime)
        yield prime
def factorize(n):
    for prime in primes():
        if prime > n: return
        exponent = 0
        while n % prime == 0:
            exponent, n = exponent + 1, n / prime
        if exponent != 0:
            yield prime, exponent

# OP's first attempt

def totient1(n):
    counter = 0
    for i in xrange(1, n):
        if gcd(i, n) == 1:
            counter += 1
    return counter

# OP's second attempt

# I don't understand the algorithm, and just copying it yields inaccurate results

# Möbius inversion

def totient2(n):
    if n == 1: return 1
    return sum(d * mobius(n / d) for d in xrange(1, n+1) if n % d == 0)
mobius_cache = {}
def mobius(n):
    result, stack = 1, [n]
    for prime in primes():
        if n in mobius_cache:
            result = mobius_cache[n]
            break
        if n % prime == 0:
            n /= prime
            if n % prime == 0:
                result = 0
                break
            stack.append(n)
        if prime > n: break
    for n in stack[::-1]:
        mobius_cache[n] = result
        result = -result
    return -result

# traditional formula

def totient3(n):
    return int(reduce(mul, (1 - 1.0 / p for p, exp in factorize(n)), n))

# traditional formula, no division

def totient4(n):
    return reduce(mul, ((p-1) * p ** (exp-1) for p, exp in factorize(n)), 1)
```

Using this code to calculate the totients of all numbers from 1 to 9999 on my desktop, averaging over 5 runs,

- `totient1` takes forever
- `totient2` takes 10s
- `totient3` takes 1.3s
- `totient4` takes 1.3s

edited Feb 20 '10 at 21:32 answered Jun 19 '09 at 18:24

ephemient
109k 24 161 292

This is the Euler totient function, phi.

It has the exciting property of being multiplicative: if gcd(m,n) = 1 then phi(mn) = phi(m)phi(n). And phi is easy to calculate for powers of primes, since everything below them is coprime except for the multiples of smaller powers of the same prime.

Obviously factorization is still not a trivial problem, but even sqrt(n) trial divisions (enough to find all the prime factors) beats the heck out of n-1 applications of Euclid's algorithm.

If you memoize, you can reduce the average cost of computing a whole lot of them.

edited Jun 19 '09 at 17:25 answered Jun 19 '09 at 17:11
Steve Jessop
191k 19 276 523

Here's a simple, straightforward implementation of the formula given on wikipedia's page, using gmpy for easy factorization (I'm biased, but you probably want gmpy if you care about playing with fun integer stuff in Python...;-):

```
import gmpy

def prime_factors(x):
    prime = gmpy.mpz(2)
    x = gmpy.mpz(x)
    factors = {}
    while x >= prime:
        newx, mult = x.remove(prime)
        if mult:
            factors[prime] = mult
            x = newx
        prime = prime.next_prime()
    return factors

def euler_phi(x):
    fac = prime_factors(x)
    result = 1
    for factor in fac:
      result *= (factor-1) * (factor**(fac[factor]-1))
    return result
```

For example, on my modest workstation, computing euler_phi(123456789) [for which I get 82260072] takes 937 microseconds (with Python 2.5; 897 with 2.4), which seems to be quite a reasonable performance.

answered Jun 19 '09 at 18:23
Alex Martelli
435k 72 807 1110

Alex, on my fairly new machine with Python 2.5 on Windows XP, your prime_factors takes 5.5 seconds on N = 2389 * 5640689, compared to a negligent time with a pure Python factorizer that goes over primes. Why? – Eli Bendersky Jul 11 '09 at 12:29

The bulk of the work is finding the primes (the prime.next_prime() call in my answer's code) -- if you already know all primes of interest, then of course the code is faster... but what do you do the first time you explore a number with a larger prime factor than the primes you have hard-coded?-) – Alex Martelli Jul 11 '09 at 21:26

Here are some links to other discussions on this- including some other language implementations:

http://www.velocityreviews.com/forums/t459467-computing-eulers-totient-function.html

http://www.google.com/codesearch?q=Euler%27s+totient&hl=en&btnG=Code

answered Jun 19 '09 at 20:13
iwanttoprogram
281 1 11