Computing Euler's Totient Function

Vishnu Raveendran

July 28, 2013

Abstract

In this paper, we discuss ways to compute Euler's totient function using the Python programming language. We show the use of the Euclidean algorithm and the Sieve of Eratosthenes as a basis for writing a function that computes totients. First, we detail a brute-force approach, along with some optimizations for such an approach. Next, we explain another approach by sieving and prime factorizations. Finally, we compare and discuss the results for the two approaches.

1 Euler's Totient Function

The totient function $\phi(n)$ counts the number of positive integers less than or equal to n that are relatively prime with n. Two integers are relatively prime [1] when their greatest common divisor is 1. This also means that the two integers have no common factors in their respective prime factorizations.

2 The GCD Approach

2.1 Introduction to the GCD Approach

We now have a way to test whether two integers a and b are relatively prime: checking that $gcd(a, b) = 1$. The Euclidean algorithm for finding the greatest common divisor can be implemented in Python as such:

```
def gcd(a, b):
    if not b:
        return a
    return gcd(b, a % b)
```

The gcd approach would be to check the greatest common divisor of every positive integer k less than n.

```
def gcd_phi(n):
    r = 1
    for x in xrange(2, n):
        if gcd(x,n) == 1:
            r += 1
    return r
```

We begin by initializing a counting variable `r` to 1 because the greatest common divisor of any n and 1 is 1. So, it is guaranteed that $\phi(n) \geq 1$. Next, we begin iterating with an iteration variable `x` from 2 to $n-1$. The upper bound is $n-1$ because n is not relatively prime with n for $n > 1$. Now, if the greatest common divisor of x and n is 1, then we increment `r` by 1. After we have exhausted the range, we return the result `r`.

2.2 Efficiency of GCD Approach

Computing $\phi(n)$ using the above gcd approach requires $n-2$ function calls to *gcd*(n). The Euclidean algorithm for computing the gcd requires $O(\log(a)\log(b))$ operations [2]. This Big-O notation just means that the most number of operations required will never be greater than product of the logarithms of a and b. We see that computing $\phi(n)$ requires $O(n\log(n)\log(b))$ operations, since n is always an input of the gcd function and we compute the gcd about n times. An example runtime of this function with a 7-digit input follows. I wrote my own timing function `time_gcd_phi(n)` which just calls `gcd_phi(n)` and measures the elapsed time using the standard Python `time` module.

```
import time
def time_gcd_phi(n):
    t = time.clock()
    print gcd_phi(n)
    print str(time.clock()-t)+‘ seconds’

>>> time_gcd_phi(1234567)
```

```
1224720
3.25122116689 seconds
```

This example takes about 3.25 seconds to run. We will compare this with a different method later.

2.3 GCD Approach Optimization

Sometimes, it is not necessary to check all x less than n. Consider the case when n is even. This means that n has 2 as a factor. In fact, every even number has 2 as a factor, so it is not possible for two even numbers to be relatively prime; their greatest common divisor would be at least 2. Therefore, when n is even, we need only check every odd number. However, when n is odd, we must still check every number.

```
def opt_phi(n):
    r = 1
    if not n % 2:
        for x in xrange(3, n, 2):
            if gcd(x,n) == 1:
                r += 1
    else:
        for x in xrange(2, n):
            if gcd(x,n) == 1:
                r += 1
    return r
```

Although the optimization only occurs when n is even, if we need to calculate $\phi(n)$ for a range of numbers, we see that this optimization would occur for half the range. By knowing whether or not n is even, we can eliminate checking all even numbers. So, if we know the prime factorization of n, we could compute how many numbers cannot be relatively prime with n as well.

3 Sieving and Prime Factorization

3.1 Prime Sieving

Let us approach the totient function using the prime factorization of n. For simplicity, we will use a trial division algorithm to find the prime factors of n. This means that we need to generate a list of primes that are possible divisors of n. This can be achieved with a Sieve of Eratosthenes.

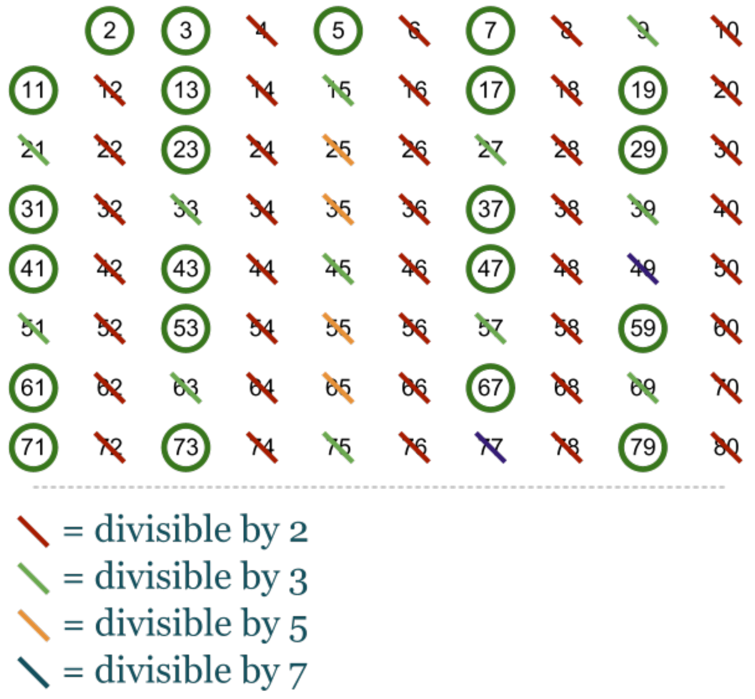

Figure 1: Sieve of Eratosthenes [3] for primes less than 100.

A prime sieve begins by eliminating multiples of 2 within a range of numbers. After the range is exhausted, the next number in the list is prime. The algorithm continues by eliminating multiples of the most recent prime and ends when the list contains only prime numbers [4]. A Python implementation follows:

```
def sieve(n):
    sieve = range(n+1)
    sieve[1] = 0
    for i in range(2, int(n**.5)+1):
        if sieve[i]:
           m = n/i - i
           sieve[i*i: n+1:i] = [0] * (m+1)
    return [x for x in sieve if x]
```

We begin by initializing a list[1] of consecutive integers, `sieve`, from 0 to n. We set the second element of the list, `sieve[1]`, to 0, since 1 is not prime. We iterate with iteration variable i from 2 to $\sqrt{n}$. A more detailed explanation of the $\sqrt{n}$ upper bound follows later. For illustrative purposes, let $n \gg 4$.

The iteration starts with $i = 2$. We see that `sieve[2]` initially holds the value 2, so the condition[2] `if sieve[i]:` is `True`. Now we calculate m, the number of times 2 goes into $n - i^2$. We only check the divisibility of $n - 4$ by 2 because we only want to eliminate multiples of 2 beginning with 4.The statement

`sieve[i*i: n+1:i] = [0]*(m+1)`

uses Python list splice syntax. It means all elements in the list `sieve` starting at index i^2 at index increments of i are replaced with 0. Since the left hand side of the assignment is a list, the right hand side must also be a list. Hence, a list of zeroes of size $m + 1$ is assigned[3] to the spliced list of the same size.

Now, you may be wondering why the upper bound of i is $\sqrt{n}$ and why we begin elimination at index i^2. Consider a range of integers from 0 to 20. Starting at 2, we eliminate multiples of 2, but not 2 itself. The next number that remains is 3, a prime. Now we eliminate multiples of 3, but not 3 itself. We note that 6 was previously eliminated as the third multiple of 2. Even in a normal[4] factorization the square of a prime p can only be written as p with multiplicity 2. So, p^2 cannot be represented as kp_m for some natural number k and some prime $p_m < p$. It then follows that we need only increment i up to $\sqrt{n}$ because the first valid elimination for a given iteration would begin at index i^2.

We continue iterating and when the list element at index i is non-zero, we know that i is prime and we begin the elimination process. We replace multiples of primes with zeroes so that all the non-zero numbers left in the list at the end of

[1]In Python, `range(n)` does not include n. We must use `range(n+1)` to initialize a list of integers from 0 to n inclusive.

[2]The condition `if sieve[i]:` is a check for holding a non-zero or not-null value.

[3]All assignments to the spliced list will affect the original list.

[4]A factorization that is not restricted to prime numbers.

the iterations are prime. We return the list of only prime numbers less than or equal to n by `return [x for x in sieve if x]` which removes all zeroes from the original list.

3.2 Prime Factorization

Now that we have a way of generating a list of primes, we can write a trial division function to give us the prime factorization of a number n.

```
def pfactorize(n, primes):
    if n in [0, 1]:
        return []
    if n in primes:
        return [n]

    R = []
    for p in primes:
        temp = n
        if n in primes:
            R.append(n)
            return R
        while not n % p:
            n /= p
        if not temp == n:
            R.append(p)
        if n == 1:
            return R
    R.append(n)
    return R
```

We begin with our inputs `n` and `primes`. The input `n` is the number for which we will find a prime factorization. The input `primes` is a list of primes required for trial division. It can be shown that the prime factorization of a number can be achieved by trial division of primes less than or equal to $\sqrt{n}$. We first get rid of trivial cases in which `n` is either in {0,1} or is a prime. In the former case, the prime factorization is an empty list: `return []`. In the latter case, the prime factorization is a list containing only `n` itself. In non-trivial cases, we need to initialize our result `R` list as an empty list.

Now we begin the trial division. We iterate[5] through `primes` and store the value of the current element in a variable `p`. We create a temporary variable `temp` and give it the value of `n`. As an example, let us assume that `n` is not in our list of primes. In such a case, we then check if `p` divides `n` with the statement `while not n % p:` and update the value of `n` with the division of `n` by `p`. This while loop divides out the multiplicity of the current prime. Then the condition `if not temp == n:` only adds `p` once to our result list `R` if division was performed since the `temp` assignment.

The iterations continue until `n` is in `primes`, `n == 1`, or we have exhausted `primes`. In the first case when `n` is in `primes`, we just add `n` to `R` and return `R`, the reduced[6] prime factorization of N[7]. In the second case when `n == 1`, we just return `R`. In the third case when we have exhausted `primes`, we also add `n` to `R` and return. The difference between the first and third case is that `n` is not in `primes` in the third case. Since no prime `p` in `primes` could divide `n`, it follows that `n` is a prime and in the prime factorization of N.

The prime factorization is now complete and we can use it to calculate $\phi(n)$.

4 Prime Factors Approach

4.1 Introduction to the Prime Factors Approach

In this section, we show how the prime factorization of n can be used to compute $\phi(n)$. We need to go through each prime in the reduced prime factorization of n and determine how many numbers in the range $[1, n]$ are multiples of p. Those numbers then cannot be relatively prime with n, as they have some prime p as a common factor. We do not need to worry about the multiplicity of a prime in the prime factorization of n. Multiplicity gives no extra information about how many numbers in our range are not relatively prime with n. Consider the number 32, which has a prime factorization of 2 with multiplicity 5. We see that all even numbers in the range $[1, 32]$ cannot be relatively prime with 32. We then conclude without even considering the multiplicity, that $\phi(n) = 16$, the number of odd numbers in the range $[1, 32]$.

In the previous example when $n = 32$, the only prime in the prime factorization of n was 2. So, we have $\frac{n}{2}$ multiples of 2 in the range $[1, n]$. It follows that we have $n - \frac{n}{2}$

[5]This Python syntax is equivalent to iteration by index followed by variable assignment.
[6]Ignoring multiplicities greater than 1.
[7]Here, N refers to the original value of `n` passed into the function.

numbers that are not multiples of 2. In general, we have $\frac{n}{p}$ multiples of a prime p and $n - \frac{n}{p}$ non-multiples of p in the range $[1, n]$. We can rewrite this as $n(1 - \frac{1}{p})$.

Now, we can think of $(1 - \frac{1}{p})$ as the probability that a number is not divisible by p in some arbitrary range $[1, n]$. When we consider the set P of all primes in the prime factorization of n, the product of all the probability terms represents the probability that a number is not divisible by any of the primes in P. After multiplying this combined probability by the arbitrary range $[1, n]$ consisting of n numbers, we have $\phi(n) = n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2})(1 - \frac{1}{p_3})...(1 - \frac{1}{p_k})$.

4.2 Python Implementation

```
def phi_fact(n, primes):
    pfactors = pfactorize(n, primes)
    r = n
    for p in pfactors:
        r *= (1-1./p)
    return int(r)
```

The inputs are once again `n` and `primes`. We first determine the prime factors of n by calling our prime factorization function `pfactorize(n, primes)` and store it in the variable `pfactors`. Next, we assign `r` the value of `n`. We then iterate through every prime in `pfactors` and update the value of `r` by the product[8] of `r` and $1 - \frac{1}{p}$. After we exhaust the list of prime factors, we return the integer value of `r`. Since we performed floating-point division in a calculation, the type of `r` changed from an integer to a floating point number. However, since all the primes in the prime factorization of n divide n, it follows that the result of our computation will be an integer. So, we can just change the type of `r` back to an integer and be done.

4.3 Efficiency of the Prime Factors Approach

We will use a similar timing function and the same n as before. This method requires us to supply a sufficient list of primes, which is a list of all primes under $\sqrt{n}$. The time measured will then represent the time required to generate the primes as well as calculate $\phi(n)$. In practice, however, we would first generate a large list of primes only once, and then we could compute the totients for many different n.

[8]In the product `r *= (1-1./p)`, the decimal point in the numerator indicates floating-point division, as opposed to integer division.

```
import time
def time_phi_fact(n):
    t = time.clock()
    primes = sieve(int(n**.5))
    print phi_fact(n, primes)
    print str(time.clock()-t)+‘ seconds’

>>> time_phi_fact(1234567)
1224720
0.0255748610127 seconds
```

The result for `phi_fact(1234567)` is the same that of `gcd_phi(1234567)`, yet our new function runs in less than 0.1 seconds. If we calculate a large list of primes beforehand, the prime factors approach would be even faster.

5 Comparison

From section 2.2, we saw that `gcd_phi(n)` took about 3.25 seconds to complete. The disparity in run-time is not so suprising when we consider the differences in the two approaches. In the gcd approach, we needed to check all natural numbers k less than n and increment the totient if k was relatively prime with n. In the prime factorization approach, we provide a list of all primes under $\sqrt{n}$. The trial division in the worst case[9] would exhaust the entire list of primes before yielding the prime factorization of n. Yet, even in the worst case there would be less than $\sqrt{n}$ iterations, since there are less primes than natural numbers in the range $[1, \sqrt{n}]$. After we have a prime factorization, our function `phi_fact(n)` simply performs a multiplication of terms. So the function run-time grows linearly with the number of unique prime factors of n. We see that the limiting step in computing $\phi(n)$ is the prime factorization of n. Although we used trial division for prime factorization, there are more efficient ways to find prime factors. These different prime factorization methods are beyond the scope of this paper, but show that calculating $\phi(n)$ can be more efficient with better prime factorization algorithms.

[9]The worst case is when n is prime.

References

[1] J. P. D'Angelo, D. B. West.*Mathematical Thinking: Problem Solving and Proofs.* 2nd ed., Prentice Hall (2000). pp. 124-127.

[2] Paul E. Black. Euclid's algorithm, *Dictionary of Algorithms and Data Structures [online].* U.S. National Institute of Standards and Technology. 14 May 2007. (accessed 15 July 2013)

URL:http://www.nist.gov/dads/HTML/euclidalgo.html

[3] Sieve of Eratosthenes

URL:http://www.eandbsoftware.org/wp-content/uploads/2013/05/SieveofEratosthenes.png

[4] B.M. Bredikhin (originator). Eratosthenes, sieve of., *Encyclopedia of Mathematics.* 7 February 2011. (accessed 15 July 2013)

URL:http://www.encyclopediaofmath.org/index.php?title=Eratosthenes,_sieve_of&oldid=17311