Calculating the Greatest Common Divisor

The *unique factorisation theorem* says that a positive integer m can be written as a product of prime powers in the form $m = p_1^{a_1} \cdots p_r^{a_r}$ where a_i are positive integers and $p_1 < \cdots < p_r$ are distinct primes; moreover, such an expression is unique.

Given another positive integer n, we can write it as $n = q_1^{b_1} \cdots q_s^{b_s}$ using the same result.

Now, let $t_1 < \cdots < t_u$ be the elements of the union $\{p_1, \ldots, p_r\} \cup \{q_1, \ldots, q_s\}$ arranged in increasing order. Allowing powers to be 0, we can use the above expressions to write:

$$m = t_1^{a'_1} \cdots t_u^{a'_u}$$
$$n = t_1^{b'_1} \cdots t_u^{b'_u}$$

for some non-negative integers a'_i and b'_i.

In this situation, the Greatest Common Divisor $\gcd(m, n)$ and the Least Common Multiple $\mathrm{lcm}(m, n)$ are defined as:

$$\gcd(m, n) = t_1^{\min\{a'_1, b'_1\}} \cdots t_u^{\min\{a'_u, b'_u\}}$$
$$\mathrm{lcm}(m, n) = t_1^{\max\{a'_1, b'_1\}} \cdots t_u^{\max\{a'_u, b'_u\}}$$

At first sight, it appears as if computation of these numbers *requires* factorisation and is thus a complicated process.

However, it turns out that the following algorithm quickly produces the greatest common divisor of m and n.

Note: The algorithm uses integer division: Given positive integers m and n such that $n > 0$, we can find *unique* integers q and r such that $m = n \cdot q + r$ and $0 \leq r < n$. Implementing integer division for multi-digit numbers is a tedious task, so we will not do so! Instead, we will use built-in integer division to the extent available.

define `gcd`(m, n):
  if $m < n$:
    return `gcd`(n, m)
  else if $n > 0$:
    Use integer division to find q and r such that $m = n \cdot q + r$ and $0 \leq r < n$.
    return `gcd`(n, r)
  else:
    return m

Using this we can also calculate the least common multiple of m and n.

define `lcm`(m, n):
  $g =$ `gcd`(m, n)

$p = m/g$
return $p \cdot n$

It is natural to ask for a *proof* that the above algorithms are correct! Of course, even though it *looks* faster than the factorisation method, we would like to know how fast it actually is!

Correctness

We make the following assertions:

- $\gcd(n, m) = \gcd(m, n)$.
- If $m = nq + r$, then $\gcd(n, r) = \gcd(m, n)$.
- If $n = 0$, then $\gcd(m, n) = m$.

All three assertions are quite easily proved. These can be called the "recursive invariance" of the given algorithm. They say that the recursive calls to `gcd` in our algorithm give the "correct" answer. Since recursive calls are often implemented via "looping back", the phrase "loop invariance" is also used.

Secondly, we note that:

> The pair (n, r) is smaller than the pair (m, n).

Since the new pair also consists of non-negative numbers and is smaller, the process *must* terminate in finitely many steps. This is Fermat's idea of *descent*: a sequence of positive integers that is decreasing must be finite!

A looping approach

A different way to implement the same algorithm is using a "while loop".

define `gcd`(m, n):
  Put $m' = m$ *and* $n' = n$ *to avoid changing the values of* m *and* n.
  Interchange m' *and* n' *if required so that* $m' \geq n'$ *holds.*
  while $n' > 0$:
    Use integer division to find q *and* r *such that* $m' = q \cdot n' + r$ *and* $0 \leq r < n'$.
    Replace m' *by* n' *and* n' *by* r.
  return m'

We now observe a property of m' and n' that holds at the start and end of every iteration of the "while loop". This property is that:

$$m' = c \cdot m + d \cdot n$$
$$n' = e \cdot m + f \cdot n$$

for suitable integers c, d, e and f.

Suppose that m_0 and n_0 are the values of m' and n' respectively when we start execution of one iteration of the loop and m_1 and n_1 are the values after

executing the calculations in that iteration. If we have:

$$m_0 = c_0 \cdot m + d_0 \cdot n$$
$$n_0 = e_0 \cdot m + f_0 \cdot n$$

then we see that

$$m_1 = c_1 \cdot m + d_1 \cdot n$$
$$n_1 = e_1 \cdot m + f_1 \cdot n$$

where $c_1 = e_0$, $d_1 = f_0$, $e_1 = c_0 - q \cdot e_0$ and $f_1 = d_0 - q \cdot f_0$. It is easier to see this in terms of matrix multiplication! The first set of equations give us:

$$\begin{pmatrix} m_0 \\ n_0 \end{pmatrix} = \begin{pmatrix} c_0 & d_0 \\ e_0 & f_0 \end{pmatrix} \cdot \begin{pmatrix} m \\ n \end{pmatrix}$$

The conditions $(m_1, n_1) = (n_0, r)$ where $r = m_0 - qn_0$ can be written as

$$\begin{pmatrix} m_1 \\ n_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} \cdot \begin{pmatrix} m_0 \\ n_0 \end{pmatrix}$$

It follows that

$$\begin{pmatrix} c_1 & d_1 \\ e_1 & f_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} \cdot \begin{pmatrix} c_0 & d_0 \\ e_0 & f_0 \end{pmatrix}$$

(Note in passing that "sophisticated mathematics" makes for "shorter explanations"!)

So what we notice is a *property* of the pairs (m', n') that is *invariant* after one iteration of the loop. This idea of *loop invariant properties* is important for various algorithms that we shall see later as well.

As before the new pair also consists of non-negative numbers and is smaller, the process *must* terminate in finitely many steps. We deduce:

> $\gcd(m, n)$ is the smallest positive integer g that can be written in the form $g = c \cdot m + d \cdot n$.

More sophisticated algorithm

Since it is not too difficult to do, we shall modify the algorithm to *also* calculate c, d, e and f so that we can write the required expression $g = cm + dn$ at the end!

define `gcdlcm`(m, n):
    Put $m' = m$ *and* $n' = n$ *to avoid changing the values of* m *and* n.
    Initialise $(c, d, e, f) = (1, 0, 0, 1)$.
    if $m' < n'$:
        Interchange m' *and* n'.
        Initialise $(c, d, e, f) = (0, 1, 1, 0)$.

while $n' > 0$:
Write $m' = qn' + r$ *using integer division.*
Replace c *by* e, d *by* f, e *by* $c - qe$, f *by* $d - qf$.
Replace a' *by* b', b' *by* r.
return a', $|e \cdot m|$, c, d

By extending the arguments above a little bit, one can prove that the return values are: $\gcd(m, n)$, $\text{lcm}(m, n)$, c and d such that $\gcd(m, n) = cm + dn$.

Time Complexity

We can examine the time complexity of the above algorithms by assuming that the *only* operation that uses significant amount of time is "integer division". This is not an unreasonable hypothesis since among all arithmetic operations, division takes the most time.

It is worth noting that this is equivalent to counting the number of iterations of the "while loop" which is also the main place where the algorithm can take time.

Each iteration of the while loop *reduces* m' to approximately m'/q. Thus, the larger the value of q is at a given iteration, the *faster* that m' reduces and thus, the quicker that the algorithm terminates!

Since $m' \geq n'$, we see that $q \geq 1$. We thus see that the algorithm takes the *most* iterations if $q = 1$ at *every* iteration. This "worst case analysis" gives us the equation:

$$\begin{pmatrix} g \\ 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}^k \cdot \begin{pmatrix} m \\ n \end{pmatrix}$$

where k is the number of iterations taken by the algorithm in the worst case.

Clearly, we can divide m and n by g to get the same number of iterations for a smaller pair of numbers.

> The smallest pair (m_k, n_k) of numbers that *need* k iterations for the algorithm to terminate satisfy the equation:
>
> $$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}^k \cdot \begin{pmatrix} m_k \\ n_k \end{pmatrix}$$

Using the fact that

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

we see that we have

$$\begin{pmatrix} m_k \\ n_k \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

We see that this means that $n_{k+1} = m_k$ and $m_{k+1} = n_k + m_k$. Combining this with $m_0 = 1$ and $n_0 = 0$, We see that the sequence (m_k) is:

$$1, 1, 2 = 1 + 1, 3 = 2 + 1, 5 = 3 + 2, \cdots$$

In other words, m_k is the $(k+1)$-th Fibonacci number F_{k+1} defined by:

- $F_0 = 0$, $F_1 = 1$, and
- $F_{k+2} = F_{k+1} + F_k$ for $k \geq 0$

We conclude that:

> The smallest pair for which the algorithm takes k iterations is (F_{k+1}, F_k), where F_k is the k-th Fibonacci number.

We note that F_k's are positive and non-decreasing. It follows that $F_{k+2} \geq 2F_k$. One concludes that F_k is $\Theta(2^{k/2})$, so it grows exponentially with k.

The input (F_{k+1}, F_k) when expressed in decimal (or binary, or any other base) has size a constant multiple of $\log(F_k)$ which is $\Theta(k)$.

> In the worst case, the algorithm takes k iterations for input of size $\Theta(k)$.

Equivalently, using the symmetry of Θ, we see that:

> The algorithm takes $\Theta(n)$ iterations for input of size n, in the worst case.

Note that each iteration requires integer division which *also* takes time depending on the size of the input. So if integer division for input of size n has time complexity $O(f(n))$, then the worst case time complexity of the GCD algorithm is $O(n \cdot f(n))$.

One can show that, like multiplication, the school method of long division has a time complexity of $O(n^2)$.It follows that using this school method of long division the time complexity of the GCD algorithm is $O(n^3)$. A slightly more precise analysis that uses the fact that after each iteration, the size of the integer division input *reduces* can be used to conclude a time complexity of $O(n^2)$.

The "Binary GCD" algorithm that exploits easy division by 2 on modern computers can be shown to have time complexity $O(n^2)$.