GCD and modular arithmetic

Yuval Filmus

May 4, 2017

In this short note, we discuss constructive and non-constructive aspects of the Euclidean algorithm. As an application, we explain how to perform modular arithmetic.

1 GCD

Let a, b be two positive integers. A positive integer g is said to be their *greatest common divisor*[1] if:

- g is a common divisor of a, b, that is, $g \mid a$ and $g \mid b$.
- If h is another common divisor of a, b then $g \mid h$.

A greatest common divisor is always unique[2] (since if g, h are two GCDs then $g \mid h$ and $h \mid g$). We can show that the GCD exists using the fundamental theorem of arithmetic: if $p_1, \ldots, p_n$ are all the primes dividing a, b and $a = \prod_{i=1}^n p_i^{\alpha_i}$, $b = \prod_{i=1}^n p_i^{\beta_i}$, then it is not hard to check that $g = \prod_{i=1}^n p_i^{\min(\alpha_i, \beta_i)}$ is a greatest common divisor of a, b. We denote the GCD of a, b by (a, b).

The proof we've just seen gives an algorithm for computing (a, b). However, this algorithm is very slow, since it requires factoring a and b. Euclid supposedly came up with a much better algorithm, known today as the Euclidean algorithm:

- Arrange a, b so that $a \geq b$.
- While $b > 0$, do $a, b \leftarrow b, a \bmod b$.
- Return a.

It is an elementary exercise to show that this algorithm always terminates, and furthermore returns the GCD (we have to extend the definition of GCD to the case in which one of a, b is zero; in this case $(a, 0) = a$). How good is this algorithm? It turns out that the worst case is given by Fibonacci numbers (defined by the recurrence $F_0 = 0$, $F_1 = 1$, and $F_{n+1} = F_n + F_{n-1}$).

Theorem 1. *If* $\max(a, b) \leq F_m$ *then the algorithm terminates after at most* m *iterations of the loop.*

Proof. We can assume that $a \geq b$. The proof is by induction on m. For $m = 0$, the loop terminates immediately. For $m = 1$, if $a \leq 1$ then the loop terminates after at most one iteration. Suppose now that $m \geq 2$. Denote by a', b' the values of a, b after the first iteration of the loop, and by a'', b'' their values after the second iteration. If $b \leq F_{m-1}$ then $a' \leq F_{m-1}$ and so induction shows that after the first iteration, the loop terminates after at most $m - 1$ further iterations, for a total of at most m iterations. If $b \geq F_{m-1}$ then

$$a'' = b' = a \bmod b = a - b \leq F_m - F_{m-1} = F_{m-2},$$

and so after the first two iterations, the loop terminates after at most $m - 2$ further iterations, for a total of at most m iterations. $\square$

[1] The second property below states that g is *greatest* with respect to the order $x \mid y$ rather than with respect to $x < y$. This is important since in other situations, such as polynomials, there is no natural total order on elements.

[2] In more general situations, all we can say is that g, h are *associates*, that is, $g = xh$ for some invertible x. The only invertible integers are ± 1, so since our GCDs are positive, they are unique.

We usually measure the running time of algorithms in terms of the input length, which in this case is roughly $n = \log_2 a + \log_2 b$. It is known that $F_m \approx \varphi^m/\sqrt{5}$, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio (in fact F_m is the rounding of $\varphi^m/\sqrt{5}$), and so $a \leq 2^n \leq F_m$ for $m = \lceil \log_\varphi 2 \rceil n$. This shows that the algorithm makes $O(n)$ iterations. Since $a \bmod b$ can be computed in polynomial time (in fact, $O(n^2)$), the entire algorithm runs in polynomial time (in fact, $O(n^3)$).

2 Bezout's identity

Bezout's identity states that for any two positive integers a, b there exist integers x, y (not necessarily positive) such that

$$xa + yb = (a, b).$$

One proof goes as follows. Let S consist of all values of $xa + yb$ (for arbitrary integers x, y) which are positive integers. Since $a, b \in S$, the set S is non-empty, and so contains a minimum $g = xa + yb$. We claim that $g = (a, b)$. Indeed, if h is a common divisor of a, b then $h \mid xa + yb = g$. It remains to show that $g \mid a$ (the same argument shows that $g \mid b$). If not, then let $h = a \bmod g > 0$, so that $a = kg + h$. Then $h = a - kg = (1 - x)a - yb \in S$ while $0 < h < g$, contradicting the definition of g. This contradiction shows that $g \mid a$, completing the proof.

While this shows the existence of integers x, y satisfying Bezout's identity, it is not clear how to find x, y using this proof. Indeed, a priori we would have to try infinitely many x, y! (Using the identity $ba - ab = 0$ we can fix that.) A better approach is to modify the Euclidean algorithm. The Euclidean algorithm is based on the identity

$$(a, b) = (b, a \bmod b).$$

Calculating (a, b) is reduced to calculating $(b, a \bmod b)$, with base cases $(a, 0) = a$. We can prove Bezout's identity using the same approach:

- Base case: $(a, 0) = a = 1 \cdot a + 0 \cdot 0$.
- Induction: Suppose that $(b, a \bmod b) = xb + y(a \bmod b)$. Let $a = sb + (a \bmod b)$. Then $(a, b) = (b, a \bmod b) = xb + y(a \bmod b) = xb + y(a - sb) = ya + (x - ys)b$.

This approach can be turned into an algorithm which, given a, b, computes (a, b) as well as integers x, y such that $xa + yb = (a, b)$, and it allows us to obtain bounds on x, y.

Theorem 2. *The integers x, y produced by the algorithm satisfy*

$$|x| \leq \frac{b}{(a, b)}, \quad |y| \leq \frac{a}{(a, b)}.$$

Proof. The proof is by induction. The base case is when $b \mid a$ (when $b = 0$ the inequality $|x| \leq \frac{b}{(a,b)}$ is violated). In this case $(b, a \bmod b) = xb + y(a \bmod b)$ for $x, y = 1, 0$, and so the new combination x', y' is $y, x - ys = 0, 1$, which clearly satisfies the inequalities. For the inductive case, suppose that $(b, a \bmod b) = xb + y(a \bmod b)$, where $|x| \leq \frac{a \bmod b}{(a,b)}$ and $|y| \leq \frac{b}{(a,b)}$. The new linear combination calculated by the algorithm is $(a, b) = x'a + y'b$, where $x', y' = y, x - ys$. We have $|x'| = |y| \leq \frac{b}{(a,b)}$ and

$$|y'| \leq |x| + s|y| \leq \frac{a \bmod b + sb}{(a, b)} = \frac{a}{(a, b)}. \qquad \square$$

This theorem also shows that the algorithm runs in polynomial time. How does this algorithm look?

- Start at $a_0, b_0 = a, b$.
- Compute $a_1, b_1 = b_0, a_0 \bmod b_0$.

- Compute $a_2, b_2 = b_1, a_1 \bmod b_1$.
- More steps of this kind.
- Compute $a_m, b_m = b_{m-1}, a_{m-1} \bmod b_{m-1}$, where $b_m = 0$.
- Let $x_m, y_m = 1, 0$.
- Compute $x_{m-1}, y_{m-1} = y_m, x_m - y_m s_m$, where $s_m = \lfloor a_m / b_m \rfloor$.
- More steps of this kind, until x_0, y_0 are computed.

This algorithm requires us to remember all the intermediate values of a_i, b_i, since we compute the x_i, y_i in reverse order. A different algorithm, the *extended Euclidean algorithm*, doesn't suffer from this problem, and thus needs to store only a constant number of values. The idea is to maintain numbers x_i, y_i, z_i, w_i so that $a_0 = x_i a_i + y_i b_i$ and $b_0 = z_i a_i + w_i b_i$. It turns out that x_i, y_i, z_i, w_i can be computed given only $x_{i-1}, y_{i-1}, z_{i-1}, w_{i-1}$ and $x_{i-2}, y_{i-2}, z_{i-2}, w_{i-2}$. It is a nice exercise to work out this algorithm.

3 Modular arithmetic

What is Bezout's identity good for? It allows us to implement modular inverse. Given $a \in \mathbb{Z}_n^*$, we can find a^{-1} (as a number in the range $1, \ldots, n-1$) by finding numbers x, y such that $xa + yn = (a, n) = 1$. This identity shows that $xa \equiv 1 \pmod{n}$, and so $x \equiv a^{-1} \pmod{n}$. While x is not guaranteed to be in the range $1, \ldots, n-1$, the remainder $x \bmod n$ is in this range.

Modular addition, subtraction and multiplication are also easy to implement. A more challenging operation is *modular exponentiation*, computing $a^b \bmod n$. Modular powers can be computed efficiently using *repeated squaring*. Repeated squaring is based on the two identities

$$a^{2x} \bmod n = (a^x)^2 \bmod n, \quad a^{2x+1} \bmod n = a(a^x)^2 \bmod n.$$

Using these identities, we can compute $a^b \bmod n$ using $O(\log b)$ modular multiplications. The reader is encouraged to work out the corresponding algorithm in complete detail.