2. Greatest common divisor

This algorithm is perhaps the most common operation that is used in computing with large integers after the basic operations. It is also the first an algorithm was written (since this algorithm was written *before* the base 10 arithmetic operations that we saw earlier). We will study three algorithms and their extensions. The fundamental question is to find the greatest common divisor or highest common factor d of a pair a, b of integers. One also knows that there is a formula $ax+by = d$ and in some applications it is necessary to determine x and y as well. A more general problem that this is to find a free basis for a subgroup of a free group (or "lattice"). We will see how to solve that problem in later sections.

2.1. **Euclid's algorithm.** The original algorithm of Euclid goes as follows. If $b = 0$ then $d = a$ and we stop. Else we compute, $a = b \cdot w + r$. We then iterate with b taking the place of a and r taking the place of b.

Since we are dealing with a, b each involving many (say N) "digits", each division takes N^2 steps as we saw earlier. Since, these divisions are iterated until we reach the GCD (which could be 1) we could iterate N times and get a total of N^3 steps. This seems too large for such a fundamental operation. So we should first get a better estimate for the number of steps and then take into account the decreasing sizes of a and b.

Lemma 3. *Let $a > b > 0$ be such that Euclid's algorithm takes exactly n steps and a is the least possible with this property. Then $a = F_{n+2}$ and $b = F_{n+1}$, where F_n defined inductively by $F_0 = F_1 = 1$ and $F_{n+1} = F_n + F_{n-1}$.*

The proof is related with *continued fractions* from which it also follows that F_n is like C^n for some constant $C > 1$. Thus it follows that we actually need only a constant multiple of $\log(N)$ steps. It can also be noted that a and b are often of the same size (especially after the first iteration). In this case w is small so the long division takes less steps than expected.

We will study the proof of the lemma when we look at real quadratic number fields.

2.2. **Binary GCD algorithm.** Using the idea that division and multiplication by powers of two is a "quick" operation for computers (based on binary arithmetic) and the fact that the difference between two odd numbers is even, we can write an algorithm for GCD that does not use division (except by 2) at all.

First of all we find the 2-adic values (say m and n) of a and b and also replace a and b by $a/2^m$ and $b/2^n$ respectively. As we saw above this can be done in time that is linear in the size of a and b. Let k denote the minimum of m and n. We now begin the iterative step.

For this iteration a and b will always be odd. First we compute $c - \rho \cdot M^p = a - b$ (where ρ is the "borrow"). If $\rho = 1$, then we replace c by its bitwise negation $\texttt{bneg}(c)$ and then add 1 (this replaces c by $M^p - c$; the sign of c is then $(-1)^\rho$). If c is zero then the GCD is $2^k \cdot a$. Otherwise, we calculate the 2-adic value l of c and replace c by $c/2^l$ so that c is odd once more. Since l is usually a smaller than $\log(M)$ this step is linear in the size of c. Now if $\rho = 1$ then we replace b by c and if $\rho = 0$ then we replace a by c and iterate. (Thus we replace a if it is larger and b if it is larger).

Note that this involves no divisions at all. Thus it can be very fast in principle. Each iteration reduces the size of the largest of a and b by at least one bit. Thus there are at most as many steps as $\log_2(a)$. Moreover, if $a = 2^N - 1$ and $b = 1$, then we can see that the process takes exactly N steps. Thus, quite often an initial division is performed in order to ensure that a and b have roughly the same size (this condition reduces the number of iterations for all GCD algorithms).

2.3. **Lehmer's Algorithm.** An alternate approach to speeding up Euclid's algorithm is due to Lehmer. One notices that when a and b have the same size, the integer part w of the quotient a/b is often single digit. Secondly, the *process* underlying Euclid's algorithm is the application of successive linear transformations (we will see this also in the Extended GCD computations later)

$$\begin{pmatrix} u \\ v \end{pmatrix} \mapsto \begin{pmatrix} A & B \\ C & D \end{pmatrix} \cdot \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} Au + Bv \\ Cu + Dv \end{pmatrix}$$

Thus, we can repeatedly try to find the w as long as it is "small" and keep track of the operations involved by means of a matrix. When we cannot proceed further, we apply this matrix to the original data, and then try again. Occasionally, we many have to break out of a "deadlock" by performing a long division.

One way to check that w is small is to compare it with the quotient a_0/b_0 of the leading digits of a and b, when a and b have the same number of digits. More specifically if the integer part of $a_0/(b_0 + 1)$ and the integer part of $(a_0 + 1)/b_0$ are equal, then they are equal to w. The detailed algorithm is given below (we assume $a \geq b$).

If b is a single digit then we apply Euclid's algorithm to get the answer. Otherwise, we set x to be the leading digit (in base M) of a and y to the the leading digit of b *at the same place* (i. e. if $a = (a_0 \dots a_p)$ then $b = (b_0 \dots b_p)$ for the *same* p). We compute the invertible matrix $\left(\begin{smallmatrix} A & B \\ C & D \end{smallmatrix}\right)$, by an iteration of the following steps after initialising it as the identity matrix.

We compute the integer quotient w_1 of $(x + A)/(y + C)$ and the integer quotient w_2 of $(x + B)/(y + D)$. We set $w = w_1$ if $w_1 = w_2$ and perform the matrix multiplication

$$\begin{pmatrix} 0 & 1 \\ 1 & -w \end{pmatrix} \cdot \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} C & D \\ A - wC & B - wD \end{pmatrix}$$

We then replace our matrix by the resulting matrix. Similarly, we replace x by y and y by $x - wy$. Then we iterate.

Lemma 4. *In the above situation at most one of $y + C$ and $y + D$ is 0. Moreover, we have the inequalities*

$$0 \leq x + A \leq M \qquad 0 \leq x + B < M$$
$$0 \leq y + C < M \qquad 0 \leq y + D \leq M$$

This ensures that all operations involved in this part of the computation are digits. This procedure will exit when $w_1 \neq w_2$ or when one of $y + C$ or $y + D$ is 0.

If the matrix computed has $B = 0$ then we need to perform a long division operation using a and b (this case occurs only when the first w_1 and w_2 and do not match). Otherwise, we replace a by $Aa + Bb$ and b by $Ca + Db$ and go back to the start.

The advantage of this procedure is that long division is only performed when absolutely essential; i. e. when the quotient w is larger than M. One can show that this case occurs sufficiently infrequently to account for the overhead of computing the matrix $\left(\begin{smallmatrix} A & B \\ C & D \end{smallmatrix}\right)$.

Proof. (of the lemma) We only need to note that at any stage x and y are the result of applying Euclid's algorithm to the original pair x and y. Similarly, $x + A$ and $y + C$ at the same stage are the result of applying Euclid's algorithm to the pair $x+1$ and y; and $x+B$ and $y+D$ are associated to the pair x and $y+1$. The lemma follows from the consequence (of Euclid's algorithm) that these are all decreasing and initially lie between 0 and M. □

2.4. **Extended GCD.** Quite often we not only need to know the GCD of two integers a and b but also to write this GCD in the form $ax+by$. In other words we need to solve the equation $ax + by = c$ when c is the GCD. In fact we can *define* the GCD as the smallest c that satisfies this equation. Equivalently, we can define the GCD as the smallest z such that there is a pair (x, z) so that $z - ax$ is divisible by b. Thus we can start with some "obvious" pairs and use them to construct new ones which are smaller.

First of all let us follow Euclid's approach. Suppose $a \geq b$. Let $u_0 = (1, a)$ and $v_0 = (0, b)$. If $b = 0$, then our solution is $(x, z) = (1, a)$. We compute, $q_0 = [a/b]$ (the integer part of a/b) and set

$$\begin{aligned} v_1 = (v_{1,1}, v_{1,2}) &= u - q \cdot v \\ u_1 = (u_{1,1}, u_{1,2}) &= v \end{aligned}$$

Proceeding inductively, if $v_{i,2} = 0$ then $(x, z) = u_i$ is a solution. Otherwise, we let $q_i = [u_{i,2}/v_{i,2}]$ and set

$$\begin{aligned} u_{i+1} = (u_{i+1,1}, u_{i+1,2}) &= v_i \\ v_{i+1} = (v_{i+1,1}, v_{i+1,2}) &= u_i - q_i \cdot v_i \end{aligned}$$

We then iterate, until $v_{i,2} = 0$, which is becoming smaller at each step. Clearly, the first step is a special case for $i = 0$ of the remaining steps[1]. At the end we take $y = z - ax/b$. Then $ax + by = z$ and z is the GCD of a and b.

Extending Lehmer's approach is very similar. As above, suppose that $a \geq b$ and let $u_0 = (1, a)$ and $v_0 = (1, b)$. We are at the step $i = 0$. If $v_{i,2} = 0$, then our solution is u_i. Now let x denote the leading "digit" of $u_{i,2}$ and y denote the leading digit of $v_{i,2}$ at the same place (as in Lehmer's algorithm). We carry through the process of Lehmer's algorithm to compute the matrix $\left(\begin{smallmatrix} A_i & B_i \\ C_i & D_i \end{smallmatrix}\right)$. If $B = 0$ then we calculate $q_i = [u_{i,2}/v_{i,2}]$ and set as above,

$$\begin{aligned} u_{i+1} = (u_{i+1,1}, u_{i+1,2}) &= v_i \\ v_{i+1} = (v_{i+1,1}, v_{i+1,2}) &= u_i - q_i \cdot v_i \end{aligned}$$

[1]One thing that needs to be stated is that we are repeatedly assigning values to variables. In the interests of efficiency, any algorithm that deals with multi-precision arithmetic should create new variables (allocate memory) and put values in them infrequently. One way to do this is to decide in advance (by *a priori* calculations) how many variables we will need and how large they can get. Then we keep track of value variable "names" by means of a symmetric matrix (the number of variables is likely to be much smaller than the numbers we are dealing with).

Otherwise ($B \neq 0$) we set

$$\begin{aligned} u_{i+1} &= Au_i + Bv_i \\ v_{i+1} &= Cu_i + Dv_i \end{aligned}$$

We then iterate the procedure until $v_{i,2}$ becomes zero, which it must since it is decreasing at each step (this argument is as in the case of Lehmer's algorithm). At the end we take $y = z - ax/b$. Then $ax + by = z$ and z is the GCD of a and b.

Finally, we turn to the binary GCD technique. As usual, if b is zero then we have the solution $(x, y, z) = (1.0, a)$; if a is zero then we have the solution $(x, y, z) = (0, 1, b)$. Otherwise, we take k to the minimum of the 2-adic values of a and b. Let $c = a/2^k$ and $d = b/2^k$. Now, if d is even, then we interchange c and d and keep track of this interchange by setting a flag f to 1.

We may now assume that we have d odd and c non-zero. We will start with the pairs $u_1 = (1, c)$ and $v_1 = (0, d)$ at $i = 1$. At each stage w_i will be a pair in which the second part is even. Thus, if c is odd we put $w_1 = u_1 - v_1$, else (if c is even) we put $w_1 = u_1$.

Now we perform the following steps inductively. First of all if $w_{i,2} = 0$ then $(x, z) = u_i$ is the required pair and we exit the induction. Next we remove powers of two from w_i. For this we take $z_{1,i} = w_i$ and induct on $z_{k,i}$ as follows.

$$z_{k+1,i} = \begin{cases} \frac{1}{2} \cdot (z_{k,i,1}, z_{k,i,2}) & \text{if } z_{k,i,1} \text{ is even} \\ \frac{1}{2} \cdot (z_{k,i,1} + d, z_{k,i,2}) & \text{if } z_{k,i,1} \text{ is odd} \end{cases}$$

Note that, by induction on k, d divides $z_{k+1,2} - cz_{k+1,1}$ in both cases. We stop as soon as $z_{k,i,2}$ is odd and put $z_i = z_{k,i}$.

The last inductive step is the re-assignment. If $z_{i,2}$ is negative, then we take $v_{i+1} = -z_i$ and $u_{i+1} = u_i$, otherwise we take $u_{i+1} = z_i$ and $v_{i+1} = v_i$. Then we put $w_{i+1} = u_{i+1} - v_{i+1}$ and induct.

When we exit the induction, we have (x, z) so that d divides $z - cx$, so let $y = z - cx/d$. If $f = 0$, then we have $ax + by = 2^k z$, while if $f = 1$ then we have $ay + bx = 2^k z$. In both cases $2^k z$ is the GCD of a and b (this follows since the steps that we are performing on the second part of the pairs u_i and v_i are exactly the same as those for binary GCD).