Computational Number Theory

1 Background material

1.1 Polynomial time

Let f, g be two functions from $\mathbb{N}^r$ to $\mathbb{R}_{\geq 0}$. We write

$$f = O(g)$$

if there exist constants A, B such that

$$f(n_1, \ldots, n_r) \leq Ag(n_1, \ldots, n_r)$$

whenever $n_1, \ldots, n_r \geq B$.

For example, the number of digits in the base b representation of a positive integer n is

$$O(\max(1, \tfrac{\log n}{\log b})).$$

We shall often be interested in estimating running times for algorithms. We may have input parameters $n_1, \ldots, n_r$, and then the running time (i,e., the number of elementary operations—what operations we consider to be elementary may depend on context) will be some function of $n_1, \ldots, n_r$, say $f(n_1, \ldots, n_r)$. A precise expression for $f(n_1, \ldots, n_r)$ may be hard to determine, and may be too complicated to provide any illumination. Often one is satisfied with finding a 'nice' function $g(n_1, \ldots, n_r)$ such that $f = O(g)$.

A frequently-met special case is where the input to an algorithm consists of a single positive integer n. Let $f(n)$ be the running time of the algorithm. If

$$f(n) = O((\log n)^d)$$

for some constant d, then we say that the algorithm runs in **polynomial time**. More generally, if b is the number of bits in the input, and there is a constant d such that an algorithm runs in time $O(b^d)$, then we say that the algorithm runs in polynomial time. If $d = 0, 1, 2, 3, \ldots$, then we say that the algorithm runs in constant time, linear time, quadratic time, cubic time,....

In practice, the implied constants A, B in the O-notation are important. A quadratic time algorithm will run more quickly than a (genuinely) cubic time algorithm for large enough values of the input, but for specific values of the input, the asymptotic behaviour is irrelevant.

Basic arithmetic operations on the integers, $+, -, \times, \div$ (finding quotient and remainder), are all polynomial-time, even when using naive methods. Hence

arithmetic in $\mathbb{Z}/n\mathbb{Z}$ can be done in polynomial time (for division, where possible, one computes a multiplicative inverse using Euclid's algorithm: see immediately below).

1.2 Euclid's Algorithm

Any positive integer n can be written uniquely (up to reordering the factors) as a product of primes. If

$$a = \prod_{i=1}^{r} p_i^{\alpha_i},$$

$$b = \prod_{i=1}^{r} p_i^{\beta_i},$$

where $p_1, \ldots, p_r$ are distinct primes, then the greatest common divisor of a and b, $\gcd(a, b)$ is given by

$$\gcd(a, b) = \prod_{i=1}^{r} p_i^{\min(\alpha_i, \beta_i)}.$$

It is the greatest integer dividing both a and b.

Computing $\gcd(a, b)$ by finding the prime factorisations of a and b can be slow (we shall be looking at this problem in some detail later). Can we do better?

If $|a| \geq |b| > 0$, then we can write

$$a = qb + r$$

with

$$|r| \leq \frac{|b|}{2}$$

(q is the nearest integer to $\frac{a}{b}$). Any common divisor of a and b divides r (and b). Any common divisor of b and r divides a (and b). Therefore $\gcd(a, b) = \gcd(b, r)$. This gives a recursive algorithm for computing $\gcd(a, b)$.

Euclid's algorithm
Input: integers a and b.
Output: $\gcd(a, b)$.

Step 1 If $|a| < |b|$, then swap a and b.

Step 2 If $b = 0$, then STOP with output $|a|$.

Step 3 Compute q, r such that $a = qb + r$, with $|r| \leq \frac{1}{2}|b|$. Replace a by b, b by r, and go to Step 2.

Each application of Step 3 reduces $|b|$ by a factor of at least 2. Hence we loop $O(\log(\min|a|, |b|))$ times, and we see that Euclid's algorithm runs in polynomial time.

There are many variants, some particularly convenient for use with binary computers. See Cohen §1.3, especially Algorithm 1.3.5.

If $d = \gcd(a, b)$, then there exist integers x, y such that $d = xa + yb$. Euclid's algorithm can be extended to find x, y. In the following, all triples (x, y, z) satisfy $xa + yb = z$.

Extended Euclid
Input: integers a and b.
Output: integers x and y such that $xa + yb = \gcd(a, b)$.

Step 1 If $|a| < |b|$, then swap a and b. Let $A = (1, 0, a), B = (0, 1, b)$.

Step 2 If $b = 0$, then STOP with output $A = (x, y, \gcd(a, b))$.

Step 3 Compute q, r such that $a = qb + r$, with $|r| \leq \frac{1}{2}|b|$. Copy A into A_{old}. Replace A by B, replace B by $A_{old} - qB$, and go to Step 2.

This algorithm can be used to compute inverses in $\mathbb{Z}/n\mathbb{Z}$ in polynomial time. Given $a \in \mathbb{Z}$, we compute x, y such that $xa+yn = \gcd(a, n)$. If $\gcd(a, n) > 1$, then a is not invertible mod n. If $\gcd(a, n) = 1$, then x is the desired inverse of a mod n.

1.3 Modular exponentiation

Computing $a^m \pmod{n}$ can be done by performing $m - 1$ multiplications $\pmod{n}$. We can do much better as follows.

Write

$$m = 2^s + 2^t + \cdots$$

where $s > t > \ldots$ (i.e., compute the binary representation of m). Then compute the list

$$a, a^2, a^4, a^8, \ldots, a^{2^s}$$

(all mod n) where each term is the square of the previous one. Multiply together all those a^{2^i} for which 2^i appears in the binary representation of m, to get

$$a^{2^s} a^{2^t} \cdots = a^{2^s + 2^t + \cdots} = a^m.$$

We have thus computed $a^m \pmod{n}$ in $O(\log m)$ multiplications.

The above idea is implemented more efficiently (without having to store either the binary expansion of m or the relevant a^{2^i}) by the following algorithm.

Modular exponentiation: $a^m \pmod{n}$

Step 1 Let $x = m, y = a, z = 1$.

Step 2 If x is odd, then multiply z by $y \pmod{n}$, and then subtract 1 from x.

Step 3 If $x = 0$, then STOP with output z.

Step 4 Square $y \pmod{n}$, divide x by 2, and go to Step 2.

See Cohen §1.2 for variants.