Number Theory

Number Theory is a branch of mathematics that explores the properties of integers, most of the time only the natural numbers. Most problems in elementary Number Theory are easy to state and understand, because they're just extensions of grade school mathematics. At the same time, the solutions to these problems are not **simple**, usually requiring ingenious insights that are beautiful and fascinating.

Take for example the problem of finding the **greatest common divisor** (gcd) of two numbers. It has many applications, the least of which is to print a fraction in reduced form (ex. 2/3 instead of 4/6). The solution is the famous Euclidean algorithm, which is simple to write, and unbelievably efficient. Learning these algorithms will enrich your understanding of the numbers that you use everyday. You may also be surprised that their applications range from magic tricks to modern day cryptography.

From now on, we assume all numbers are non-negative integers (naturals) unless otherwise noted. In grade school mathematics, we learn how to divide using **long division**, a process that finds the quotient one digit at a time, and in the end produces both the **quotient** and **remainder**. One property that we should remember is that the remainder is always less than the divisor. So we have the

Division Algorithm (Long Division)
Given a dividend a, and a divisor b, we can always divide a by b and get a = q*b + r, where q is the quotient and r is the remainder. Moreover, $0 \le r < b$.

Given a and b, the C++ operation `a/b` gives the quotient (when both are `int`), and `a%b` gives the remainder. We say that `b` divides `a` if the remainder `a%b = 0`. Now, for any two integers A and B, a common divisor d is one that divides both A and B. The **greatest common divisor** of A and B is a number g such that

(i) g is a common divisor of A and B, and
(ii) if d is any other common divisor of A and B, then $d \le g$.

How do we find the gcd? We can brute force the solution, but that is of course too slow. It turns out there is a simple algorithm, called the Euclidean algorithm, that finds the gcd very quickly. However, to prove that it works is a bit tricky.

Euclidean Algorithm

```
int gcd( int A, int B ) {
   if ( B == 0 ) return A;
   else return gcd( B, A%B );
}
```

This algorithm does two things. When B is 0, return a as the gcd. Otherwise, take `gcd(A,B) = gcd(B,A%B)`. Assuming $A \ge B$ to start with, the algorithm repeatedly calculates the following:

$$
\begin{aligned}
A &= q_1 \times B + r_1 \\
B &= q_2 \times r_1 + r_2 \\
r_1 &= q_3 \times r_2 + r_3 \\
&\vdots \\
r_{n-3} &= q_{n-1} \times r_{n-2} + r_{n-1} \\
r_{n-2} &= q_n \times r_{n-1} + r_n \\
r_{n-1} &= q_{n+1} \times r_n + 0.
\end{aligned}
$$

We recursively call the gcd function, taking the first column ($A,B,r_1,...$) as the first argument and the third column ($B,r_1,r_2,...$) as the second argument. So, we first call gcd(A,B), then gcd(B,r_1), then gcd (r_1,r_2), etc., until we reach gcd(r_n,0), which returns r_n. The Euclidean Algorithm returns $g=r_n$ as the gcd of A and B. Why is this algorithm correct, and why does it always terminate?

Proof of Termination. By the division algorithm, we get $0 \le r_1 < B$, $0 \le r_2 < r_1$, ..., so $B > r_1 > ... \ge 0$, but they are all integers, and so in at most B steps, we will get a remainder $r_n=0$ which ends the algorithm. Q.E.D.

Proof of Correctness. To prove correctness, we need to prove that $g=r_n$ is both a common divisor of A and B, and is the greatest one. First, let's build from the bottom up. From the last equation, we get r_n divides r_{n-1}. Then one line up we see that r_n must divide r_{n-2} as well, because it divides both r_{n-1} and r_n on the right. Continuing this logic, we see that r_n divides r_{n-3}, ..., until we reach the first two lines, from which we deduce that r_n divides A and B as well. Hence, $g=r_n$ is a **common divisor.**

Now, to prove that it is indeed the greatest common divisor, we take any other common divisor d that divides both A and B, and work our way top-down. From the first line, because d divides A and B, d must divide r_1. Similarly, d must then divide r_2, and r_3, etc., until we reach the last line. We conclude from this that d divides r_n as well, and so any other common divisor of A and B must also divide $g=r_n$. Hence, g is the **greatest common divisor**. Q.E.D.

We have already mentioned that the Euclidean Algorithm is fast. Before proving this mathematically, we'll first look at an example.

Q: Find gcd(307829,301337)
A: 307829 = 1 x 301337 + 6492
301337 = 46 x 6492 + 2705
6492 = 2 x 2705 + 1082
2705 = 2 x 1082 + 541 ← 541 is the gcd.
1082 = 2 x 541 + 0.

In 5 iterations we found the gcd of two pretty big numbers! Indeed, the algorithm is efficient in general, as we shall now show.

Lemma 1. Take $B=r_0 > r_1 > ... \ge 0$. Then

$$r_{i+2} < \frac{r_i}{2} \quad ,$$

and from this we conclude that the Euclidean Algorithm runs in $O(2\log_2(B))$.

Proof. Every line in the algorithm is of the form

$r_i = q_{i+2} \times r_{i+1} + r_{i+2}$, and
$r_i \ge q_{i+2} \times r_{i+2} + r_{i+2}$
$\ge (q_{i+2}+1) \times r_{i+2}$
$\ge 2 \times r_{i+2}$,

Q.E.D.

So, the algorithm is logarithmic in time, which is pretty fast. With our bound, the slowest run-time for 32-bit integers is ~64 iterations. The strongest bound known is by Lamé, who showed that in the worst case, the number of iterations is always ≤ 5 times the number of digits in the smaller number.

Extended Euclidean Algorithm

In the "real" world, simple algebra tells us that if we're given A, B, and the equation

$$A\ x = B,$$

then the solution is $x = B / A$. But, when we restrict ourselves to integers only, the story is quite different. To learn how to solve linear equations in integers, we first study a related equation.

Given two natural numbers A and B, we would like to see what kinds of numbers we can create using the equation $A\ x + B\ y$, where x and y are some other integers (can be negative). First, let $g = \gcd(A,B)$. Then, since g divides A and B, g must also divide $A\ x + B\ y$. This proves that

$$|A\ x + B\ y| \geq g,$$

which means the **smallest possible positive value** of $A\ x + B\ y$ is g. Using the Euclidean algorithm, we shall prove that it's not just possible, but we can **always** find x, y so that $A\ x + B\ y = g$.

The trick to this is to look at our "equation list" from the Euclidean algorithm above, and notice how we can use it to construct a solution while we are computing g. The idea is simple. When we iterate one level at a time, we can create numbers x_i and y_i that sums to the remainder using the equation. When we progress down to the last level, we get $g = r_n = A\ x_n + B\ y_n$, which solves the problem.

To do this, first look at the first line

$$A = q_1 \times B + r_1$$

Then obviously $r_1 = A \times 1 + B \times -q_1$, giving $x_1 = 1$, $y_1 = -q_1$. On the second line

$$B = q_2 \times r_1 + r_2$$

we get $r_2 = B \times 1 + r_1 \times -q_2$

$= B \times 1 + (A \times 1 + B \times -q_1) \times -q_2$

$= A \times -q_2 + B \times (1 + q_1 q_2)$,

giving $x_2 = -q_2$, $y_2 = (1 + q_1 q_2)$. Continuing this way, it is not hard to see that we will end with x_n and y_n eventually. But, the real implementation is from bottom-up because we use results from a recursive call to build the solution. Here is such an implementation of the **Extended Euclidean** algorithm that returns not only $g = \gcd(A,B)$, but also two integers x, y so that $A\ x + B\ y = g$.

```
// A triplet structure that stores divisor g, and x, y as above
struct Triple
{
   int g, x, y;
   Triple( int d, int w, int e ) : g( d ), x( w ), y( e ) {}
};

// Extended gcd – Returns <g,x,y>, so that g=gcd(A,B), and A*x+B*y=g
Triple egcd( int A, int B )
{
    if( B == 0 ) return Triple( A, 1, 0 );  // Base case
    Triple tr = egcd( B, A % B );           // Call next iteration
    return Triple( tr.g, tr.y, tr.x - A / B * tr.y );  // Solve
}
```

The egcd(A,B) function has a base case and a recursive case. The base case is trivial – when B=0, we return g=gcd=A, x=1, y=0. Now, when we recurse, we set 'tr' as the triple that solves gcd(B,A%B). Hence,

(1) $B * tr.x + (A\%B) * tr.y = tr.g$

Just as before, gcd(A,B) = gcd(B,A%B), so we don't need to touch the gcd. What we need now is to return a triple tr2 so that

$$A * tr2.x + B * tr2.y = tr2.g = tr.g$$

But, we know that $A = \lfloor A/B \rfloor * B + (A\%B)$, so (1) tells us that

$$B * tr.x + (A - \lfloor A/B \rfloor * B) * tr.y = tr.g$$
$$B * tr.x + A * tr.y - \lfloor A/B \rfloor * B * tr.y = tr.g$$
$$A * tr.y + B * (tr.x - \lfloor A/B \rfloor * tr.y) = tr.g$$

which gives tr2.x = tr.y, and tr2.y = $tr.x - \lfloor A/B \rfloor * tr.y$, which is precisely what we have implemented above.

Solving Ax+By=C in integers

Here is one obvious application of the extended Euclidean algorithm. Suppose we are given integers A, B and C and want to solve the equation Ax+By=C for integers x and y. x and y are allowed to be zero or negative. There could be either zero solutions (for example, 3x+9y=2) or an infinite number of solutions (2x+3y=5).

The key idea is to notice that the GCD of A and B can be written as As+Bt for some integers s and t, and as shown in the previous section, this is the smallest positive value that any linear combination of A and B can have. Moreover, if we pick any other integers s' and t' then As'+Bt' will be a multiple of the gcd(A,B). This lets us immediately solve the "no solution" case – if gcd(A,B) does not divide C, then there is no hope of finding a linear combination of A and B that will be equal to C. For example, if we want to solve 3x+9y=2, then we note that gcd(3,9)=3 does not divide 2, so there is no solution.

If gcd(A,B) does divide C, then we can take gcd(A,B)=g=As+Bt and compute the number z that we need to multiply it by to get C:

$$(Ax+Bt)z = A(sz) + B(tz) = C.$$

We have a solution: {x=sz, y=tz}. But there are more. We can add B/g to x and subtract A/g from y:

$$A(x+B/g) + B(y- A/g) = Ax + By + AB/g - AB/g = Ax + By = C.$$

This is another solution. In fact, we can add B/g to x and subtract A/g from y as many times as we want and get a new solution every time. Therefore, there is an infinite number of solutions {x',y'}, each one of the form

$$x' = x + tB/g,$$
$$y' = y - tB/g,$$

where t is any integer (positive, negative or zero). Here is an implementation of this algorithm.

```
Triple linearDiophantineEquationSolver( int A, int B, int C ) {
    Triple tr = egcd( A, B );
    if( C % tr.g != 0 ) return Triple( 0, 0, 0 );
    tr.x *= C / tr.g; t.y *= C / tr.g;
    return tr;
}
```

If the returned triple has tr.g=0, then there is no solution. Otherwise, tr.g is gcd(A,B), and {tr.x,tr.y} is one of the infinitely many solutions.