Euclid's GCD Algorithm & Modular Arithmetic

1 Euclid's GCD Algorithm:

Definition: The greatest common divisor of a and b is the largest d such that $d|a$ and $d|b$ (where $d|a$ denotes that d divides a).
Example: To compute the gcd of 360 and 84, we could just factor them into prime factors: $360 = 2^3 \times 3^2 \times 5$ $84 = 2^2 \times 3 \times 7$.

Now the gcd is just the product of the common prime factors (with multiplicity): $gcd(360, 84) = 2^2 \times 3 = 12$. In general, this method is extremely inefficient, since it starts by factoring the two numbers — a task for which we only have exponential time algorithms.

The following algorithm due to the ancient Greek mathematician Euclid provides a clever way of computing $gcd(a, b)$ without factoring a and b. Let us assume for this lecture that $a \geq b \geq 0$.

```
Function Euclid(a,b)
    if b = 0 return a
    return(Euclid(b, a(mod b)))
end Euclid.
```

Euclid's algorithm is based on the following simple fact: $gcd(a, b) = gcd(b, a(\text{mod}b))$. Prove it!

The basic step of the algorithm replaces the pair of numbers (a, b) with the *smaller* pair $(b, a(\text{mod}b))$. The key to analyzing the running time of the algorithm is bounding how much smaller $a(\text{mod}b)$ is compared to a.
Claim: $a(\text{mod}b) \leq a/2$.
Proof: Either $b \leq a/2$, in which case $a(\text{mod}b) \leq b \leq a/2$. Or $b > a/2$, in which case $a(\text{mod}b) = a - b \leq a/2$.

The number of recursive calls to the algorithm is clearly bounded by $2 \log a$, since in two recursive invocations of the algorithm, the larger number, a, is replaced by at most $a/2$. Since each call requires a division (to reduce $a(\text{mod}b)$), it requires $O(\log^2 a)$ steps, for a grand total of $O(\log^3 a)$ steps.

Extended Euclid's Algorithm

Euclid's algorithm can be extended to give not just $d = gcd(a, b)$, but also two integers x, y such that $ax + by = d$. What is the use of these extra numbers? Suppose you are not sure whether your gcd program is correct, and you wish to check the correctness of the answer. One easy test is to simply check whether d divides a and d divides b. Clearly this is not sufficient, since it only verifies that d is a common factor, not that it is the greatest common factor.

It turns out that if we check that 1. d divides a and d divides b. 2. $ax + by = d$.

Then d is necessarily the gcd of a and b. To see this, notice that if e divides a and e divides b, then e must also divide $ax + by$, and therefore e divides d. Thus d must be at least as large as every common factor e of a and b, and is therefore the gcd of a and b.

A very simple modification to the algorithm Euclid above gives us an algorithm that on input a, b returns the triple (d, x, y). these numbers x and y:

```
Function Extended-Euclid(a,b)
    if b = 0 return (a, 1, 0)
    Let a = bk + r.
```

$(d, x, y) =$ Extended-Euclid(b, r)
return$(d, y, x - ky)$
end Euclid.

We establish the correctness of the algorithm by induction on $a + b$. Now inductively assuming that the recursive step is correct (since $b + r$ is smaller than $a + b$), we have that $bx + ry = d$. But $r = a - bk$. Substituting, we get that $bx + (a - bk)y = d$. Collecting terms, we have that $ay + b(x - ky) = d$. This is exactly what the algorithm output.

Modular Arithmetic

There are two ways of thinking of modular arithmetic. The first, the naive method, is to think of doing arithmetic with just the numbers from 0 to $n - 1$. Arithmetic is almost identical to ordinary arithmetic except that whenever we obtain a number outside the interval $[0, n - 1]$ we replace it with its remainder after division by n. For example, $9 \cdot 6 \bmod 22 = 10$.

For this class, however, you may find it useful to use the following more sophisticated version of modular arithmetic. In this version, we make use of all the integers, but include the following relation: $a = b(\text{mod}n)$ if $a - b$ is divisible by n. Thus, $-15 = -8 = -1 = 6 = 13 = -8 \bmod 7$. So we may think of the first, more naive method as simply replacing a by the remainder b that results when we divide a by n. The sophisticated version simplifies many computations: for instance, $6^{1}27(mod7)$ is much easier to compute if we represent 6 as -1. Moreover, the sophisticated version is necessary for many of the proofs in the following sections.

It is easy to see that the time required to add or subtract two numbers mod$n$ is $O(\log n)$. This is because a modular addition can be carried out by adding the two numbers and then subtracting n if the result exceeds $n - 1$. Similarly multiplication of two numbers mod$n$ can be carried out by a standard multiplication followed by a division by n (to determine the remainder).

Division mod$n$ is more complicated: let us first ask the question 'does $\frac{1}{a} \bmod n$ exist? i.e. is there a number $b \bmod n$ such that $b = \frac{1}{a} \bmod n$? i.e. is there a number $b \bmod n$ such that $ab = 1 \bmod n$. If so, then dividing by a is the same as multiplying by b. For example, $d = ac \bmod n$, then to divide d by a we just multiply to get $bd \bmod n = bac \bmod n = c \bmod n$.

It turns out that $b = \frac{1}{a} \bmod n$ exists iff $gcd(a, n) = 1$. If $gcd(a, n) = d \neq 1$, then for any $b \bmod n$, $ab \bmod n$ must be divisible by d. So $\frac{1}{a} \bmod n$ does not exist in this case. On the other hand, if $gcd(a, n) = 1$, then by the extended Euclid algorithm, there are numbers x, y such that $ax + ny = 1$. Reducing this equation mod$n$, we get that $ax = 1 \bmod n$. Thus $\frac{1}{a} \bmod n$ can be computed using the extended Euclid algorithm, and can be carried out in $O(\log n)$ divisions.