University of Maryland
CMSC456 — Introduction to Cryptography
Professor Jonathan Katz

Lecture 25

1 Algorithmic Number Theory

Before turning to public-key cryptography, it will be useful to re-examine some elements of number theory. Until now, we have mainly presented number-theoretic results from a very "theoretical" point of view, rather than from a "practical" point of view. To give a specific example, we discussed in class that if p is an odd prime then exactly half the elements in $\mathbb{Z}_p^*$ are quadratic residues; but we did not really discuss how to tell when a given element is a quadratic residue nor did we discuss how to find a square root of a quadratic residue. Of course, from a purely abstract point of view the problem is (typically) easy: given a quadratic residue $y \in \mathbb{Z}_p^*$ we can always exhaustively search through all elements of $\mathbb{Z}_p^*$ until we find a correct square root. But such an answer is unsatisfying in our case, since we will only be interested in algorithms running in *polynomial time*; thus, the question becomes (e.g., in the case of the above example): how can we compute square roots in $\mathbb{Z}_p^*$ in polynomial time.

Before continuing, a word is in order regarding "polynomial time". Note that this *always* means polynomial time in the length of the input (i.e., $|p|$ in the case above) and *not* polynomial time in the magnitude of the input (i.e., p). This distinction is crucial in the case of designing efficient algorithms (note that $|p| = \lceil \log_2 p \rceil = O(\log p)$). The numbers that are used for cryptography are typically huge (say, 512 bits long) and the difference between an algorithm running in time linear in $p = 2^{512}$ and one running in time linear in $|p| = 512$ is enormous!

1.1 Simple Arithmetic Operations

Since we will be dealing with such large numbers, it is worth verifying that the "simple" arithmetic operations we are used to can actually be evaluated in polynomial time. We state the following facts without proof (in the following, a, b are positive integers with $a \geq b$):

1. Addition/subtraction of a and b can be done in time $O(|a|)$.
2. Multiplication of a and b can be done in time $O(|a| \cdot |b|)$. In fact, this can be improved to $O(|a| \cdot \log |a| \cdot \log \log |a|)$ using the discrete Fourier transform (DFT); we do not give details here.
3. Division of a by b (returning both the quotient and the remainder) can be done in time $O(|a| \cdot |b|)$. Note this means that we can compute $a \bmod b$ in time $O(|a| \cdot |b|)$. This can be also improved using the DFT.

4. Modular exponentiation (i.e., computing $a^b \bmod N$) can be done using $O(|b|)$ multiplications modulo N. Since each multiplication modulo N (including the modular reduction) can be done in time $O(|N|^2)$, this gives an $O(|b| \cdot |N|^2)$ algorithm for exponentiation. (Of course, if the complexity of multiplication is improved using the DFT then the total complexity of exponentiation also improves.) We discuss an efficient algorithm for exponentiation below.

Addition, subtraction, and multiplication can all be done using the simple, "grade-school" algorithms for these operations. But we elaborate for the case of exponentiation. A naive way of computing $a^b \bmod N$ is to compute $a^2 = a \cdot a; a^3 = a \cdot a^2, \ldots, a^b = a \cdot a^{b-1}$ and then finally reduce a^b modulo N (what is the time required for this approach?). A somewhat less naive method is to reduce the value at each step; i.e., compute $a^2 \bmod N = a \cdot a \bmod N; a^3 \bmod N = a \cdot a^2 \bmod N, \ldots$. Note, however, that this algorithm has b steps so the running time will be exponential in $|b|$!

A more careful algorithm — *repeated squaring* — is needed. Pseudocode for this algorithm follows (we assume $b > 0$ for simplicity).

```
Input: a, b, N
  if (b = 0) return 1
  ans = a, tmp = 1
  // we maintain the invariant that our solution is tmp * ans^b mod N
  while (b > 1) {
   if (b is odd) {
    tmp = tmp * ans mod N
    b = b − 1 }
   ans = ans * ans mod N
   b = b/2 }
  return (tmp * ans mod N)
```

Note that this algorithm performs at most $2|b|$ multiplications and each multiplication takes time $O(|N|^2)$ (we assume here that $a < N$ — if this is not the case we reduce a before starting). So the total running time of the algorithm is $O(|b| \cdot |N|^2)$, as claimed above.

1.2 The Euclidean and Extended Euclidean Algorithms

Very frequently, it is necessary to compute $\gcd(a, b)$ for two integers a and b. We now discuss an algorithm — the *Euclidean algorithm* — that can compute this in polynomial time.

Assume that $a > b$. The first thing to notice is that $\gcd(a, b) = \gcd(b, a - b)$ (verify this yourself). This suggests the following algorithm:

```
gcd(a, b)
 if b > a then return gcd(b, a)
 if b divides a then return b
 return gcd(b, a − b)
```

This recursive algorithm will certainly return the correct answer (the arguments to the gcd function always decrease but remain greater than 0, so the algorithm will terminate; when the algorithm terminates it always returns the correct answer). But what is the running time of this algorithm? Well, we can't really say much other than the fact that the largest input to gcd drops by $O(1)$ each time a recursive call is made. Thus, $O(a)$ recursive calls might be necessary to evaluate $\gcd(a, b)$. (In fact, the algorithm may take this long. Let a be odd and consider the computation $\gcd(a, 2)$. This will recursively evaluate $\gcd(a-2, 2), \ldots, \gcd(3, 2)$ which is roughly $a/2$ recursive evaluations.) But this is not polynomial time!

In fact, a better algorithm is possible. Instead of recursively calling $\gcd(a - b, b)$, why not subtract off as many multiples of b as we can? In other words, we may recursively call $\gcd(a \bmod b, b)$. The algorithm then becomes the following (known as the *Euclidean algorithm*):

$\gcd(a, b)$
 if b divides a then return b
 return $\gcd(b, a \bmod b)$ (note that $a \bmod b < b$)

We can bound the number of recursive calls via the following claim.

Claim *The value of the smallest argument to the* gcd *algorithm drops by at least a factor of 2 in every two recursive calls.*

Proof Let the initial call be $\gcd(a, b)$. Assuming this is not the final call (i.e., assuming that b does not divide a) the next recursive call will be $\gcd(b, a' = a \bmod b)$. If $a' \leq b/2$ then we are done. Otherwise, the next recursive call will be $\gcd(a', b - a')$. But since $a' > b/2$ then $b - a' < b/2$ and we have proved the claim. ■

The claim indicates that the algorithm makes at most $2 \log b = 2|b|$ recursive calls, and since $a \bmod b$ can be computed in time $|a| \cdot |b|$ the total running time is $O(|a| \cdot |b|^2)$ (and, in particular, the algorithm runs in polynomial time).

It is also true (we do not prove it here) that if $\gcd(a, b) = r$ then there exist two (possibly negative) integers X, Y with $|X| \leq b/2$ and $|Y| \leq a/2$ such that $Xa + Yb = r$. In the particular case when $\gcd(a, b) = 1$ we get $Xa + Yb = 1$. Computing such X and Y will be very useful (we show why below), and in fact can be done using the *Extended Euclidean algorithm.* While we do not give pseudocode for this algorithm, we hope the following example and discussion will allow the reader to develop the algorithm on their own.

Note that every time the Euclidean algorithm recurses, we evaluate $a \bmod b$. Instead of just keeping track of the remainder, we may also keep track of the number of times b divided a. Thus, besides recursively calling $\gcd(b, a \bmod b)$ we can imagine also recording the fact that $a - nb = a \bmod b$ (where $n = \lfloor \frac{a}{b} \rfloor$).

Now, If $\gcd(a, b) = 1$ then the final recursive call of the algorithm will be something like $\gcd(a', 1)$. Let the second to last call to the algorithm be $\gcd(b', a')$. Then we know that $b' \bmod a' = 1$ and moreover that

$$b' - n'a' = 1$$

for some integer n'. Let the third-to-last call be $\gcd(a'', b')$. We again know that $a'' \bmod b' = a'$ and furthermore that

$$a'' - n''b' = a'$$

for some integers n''. Here is the key to the algorithm: substituting for a' gives:

$$\begin{aligned} b' - n'(a'' - n''b') &= 1 \\ \Rightarrow (1 + n'n'')b' - n'a'' &= 1. \end{aligned}$$

Continuing in this manner (substituting each time) yields, for each recursive call $\gcd(a_i, b_i)$, integers n_i, n'_i for which $n_i a_i + n'_i b_i = 1$. Ultimately, working backward to the first (initial) function call, we get integers X and Y for which $Xa + Yb = 1$.

We conclude with an example. Consider $\gcd(55, 19)$. The Euclidean algorithm will run as follows:

$$\begin{array}{rl} \gcd(55,19) & 55 - 2 \cdot (19) = 17 \\ \gcd(19,17) & 19 - 17 = 2 \\ \gcd(17,2) & 17 - 8 \cdot (2) = 1 \\ \gcd(2,1) & \text{output } 1. \end{array}$$

Working backwards, we see that $17 - 8 \cdot (2) = 1$. Substituting for 2 gives:

$$\begin{aligned} 17 - 8 \cdot (19 - 17) &= 1 \\ \Rightarrow 9 \cdot (17) - 8 \cdot (19) = 1 \end{aligned}$$

Substituting for 17 gives: $9 \cdot (55 - 2 \cdot (19)) - 8 \cdot (19) = 1$ or:

$$9 \cdot (55) - 26 \cdot (19) = 1,$$

the desired answer.