Lecture Notes 9

41 Exponentiation: Speeding up the Computation

In section 40 (lecture notes 8), we described how to control the growth in the lengths of numbers when computing $m^e \bmod n$, for numbers m, e, and n which are 1024 bits long. Nevertheless, there is still a problem with the naive exponentiation algorithm that simply multiplies m by itself a total of $e-1$ times. Since the value of e is roughly 2^{1024}, that many iterations of the main loop would be required, and the computation would run longer than the current age of the universe (which is estimated to be 15 billion years). Assuming one loop iteration could be done in one microsecond (very optimistic seeing as each iteration requires computing a product and remainder of big numbers), only about 30×10^{12} iterations could be performed per year, and only about 450×10^{21} iterations in the lifetime of the universe. But $450 \times 10^{21} \approx 2^{79}$, far less than $e-1$.

The trick here is to use a more efficient exponentiation algorithm based on repeated squaring. To compute $m^e \bmod n$ where $e = 2^k$ is a power of two requires only k squarings, i.e., one computes

$$\begin{aligned} m_0 &= m \\ m_1 &= (m_0 * m_0) \bmod n \\ m_2 &= (m_1 * m_1) \bmod n \\ &\vdots \\ m_k &= (m_{k-1} * m_{k-1}) \bmod n. \end{aligned}$$

Clearly, each $m_i = m^{2^i} \bmod n$. m^e for values of e that are not powers of 2 can be obtained as the product modulo n of certain m_i's. In particular, express e in binary as $e = (b_s b_{s-1} \ldots b_2 b_1 b_0)_2$. Then m_i is included in the final product if and only if $b_i = 1$.

It is not necessary to perform this computation in two phases as described above. Rather, the two phases can be combined together, resulting in a slicker and simpler algorithm that does not require the explicit storage of the m_i's. I will give two versions of the resulting algorithm, a recursive version and an iterative version. I'll write both in C notation, but it should be understood that the C programs only work for numbers smaller than 2^{16}. To handle larger numbers requires the use of big number functions.

```
/* computes m^e mod n recursively */
int modexp( int m, int e, int n)
{
  int r;
  if ( e == 0 ) return 1;          /* m^0 = 1 */
  r = modexp(m*m % n, e/2, n);     /* r = (m^2)^(e/2) mod n */
  if ( (e&1) == 1 ) r = r*m % n;   /* handle case of odd e */
  return r;
}
```

This same idea can be expressed iteratively to achieve even greater efficiency.

```
/* computes m^e mod n iteratively */
int modexp( int m, int e, int n)
{
  int r = 1;
  while ( e > 0 ) {
    if ( (e&1) == 1 ) r = r*m % n;
    e /= 2;
    m = m*m % n;
  }
  return r;
}
```

The loop invariant is $e > 0 \wedge (m_0^{e_0} \bmod n = rm^e \bmod n)$, where m_0 and e_0 are the initial values of m and e, respectively. It is easily checked that this holds at the start of each iteration. If the loop exits, then $e = 0$, so r is the desired result. Termination is ensured since e gets reduced during each iteration.

Note that the last iteration of the loop computes a new value of m that is never used. A slight efficiency improvement results from restructuring the code to eliminate this unnecessary computation. Following is one way of doing so.

```
/* computes m^e mod n iteratively */
int modexp( int m, int e, int n)
{
  int r = ( (e&1) == 1 ) ? m % n : 1;
  e /= 2;
  while ( e > 0 ) {
    m = m*m % n;
    if ( (e&1) == 1 ) r = r*m % n;
    e /= 2;
  }
  return r;
}
```

42 Modular Arithmetic

In this and following sections, we review some number theory that is needed for understanding RSA. These lecture notes only provide a high-level overview. Further details are contained in course handouts 5–7 and in Chapter 5 of the textbook (Stinson).

42.1 Division theorem: quotient and remainder

Let a, b be integers and assume $b > 0$. The *division theorem* asserts that there are unique integers q (the *quotient*) and r (the *remainder*) such that $a = bq + r$ and $0 \leq r < b$. We denote the quotient by $a \div b$ and the remainder by $a \bmod b$. It follows that

$$a = b \times (a \div b) + (a \bmod b).$$

or equivalently,

$$a \bmod b = a - b \times (a \div b).$$

The latter actually defines $\bmod$ in terms of '$\div$'. The '$\div$' operator in turn can be defined as $a \div b = \lfloor a/b \rfloor$, where '/' is ordinary real division and $\lfloor x \rfloor$, the *floor* of x, is the greatest integer less than or equal to x.

When either a or b is negative, there is no consensus on the definition of $\bmod$. According to our definition, $a \bmod b$ is always in the range $[0 \ldots b-1]$, even when a is negative. For example,

$$(-5) \bmod 3 = (-5) - 3 \times ((-5) \div 3) = -5 - 3 \times (-2) = 1.$$

In the C programming language, the mod operator % is defined differently, so $(a \,\%\, b) \neq (a \bmod b)$ when a is negative and b positive.[1]

42.2 Divides

We say that b *divides* a (exactly) and write $b \,|\, a$ in case $a \bmod b = 0$.

Fact If $d \,|\, (a+b)$, then either d divides both a and b, or d divides neither of them.

To see this, suppose $d \,|\, (a+b)$ and $d \,|\, a$. Then by the division theorem, $a + b = dq_1$ and $a = dq_2$ for some integers q_1 and q_2. Substituting for a and solving for b, we get

$$b = dq_1 - dq_2 = d(q_1 - q_2).$$

But this implies $d \,|\, b$, again by the division theorem.

42.3 Modular Arithmetic

We just saw that $\bmod$ is a binary operation on integers. Mod is also used to denote a relationship on integers:

$$a \equiv b \pmod{n} \quad \text{iff} \quad n \,|\, (a-b).$$

That is, a and b have the same remainder when divided by n. An immediate consequence of this definition is that

$$a \equiv b \pmod{n} \quad \text{iff} \quad (a \bmod n) = (b \bmod n).$$

Thus, the two notions of mod aren't so different after all!

For fixed n, the resulting two-place relationship $\equiv$ is an equivalence relation. Its equivalence classes are called *residue* classes modulo n and are denoted using the square-bracket notation $[b] = \{a \mid a \equiv b \pmod{n}\}$. For example, for $n = 7$, we have $[10] = \{\ldots -11, -4, 3, 10, 17, \ldots\}$. Clearly, $[a] = [b]$ iff $a \equiv b \pmod{n}$. Thus, $[-11]$, $[-4]$, $[3]$, $[10]$, $[17]$ are all names for the same equivalence class. We choose the unique integer in the class that is in the range $[0 \ldots (n-1)]$ to be the *canonical* or preferred name for the class. Thus, the canonical name for the class containing 10 is $[10 \bmod 7] = [3]$.

The relation $\equiv \pmod{n}$ is a *congruence* relation with respect to addition, subtraction, and multiplication of integers. This means that for each of these arithmetic operations $\odot$, if $a \equiv a' \pmod{n}$ and $b \equiv b' \pmod{n}$, then $a \odot b \equiv a' \odot b' \pmod{n}$. This implies that the class containing the result of $a + b$, $a - b$, or $a \times b$ depends only on the classes to which a and b belong and not the particular representatives chosen. Hence, we can perform arithmetic on equivalence classes by operating on their names. The result class will not depend on the particular representatives chosen.

[1] For those of you who are interested, the C standard defines $a \,\%\, b$ to be the number satisfying the equation $(a/b) * b + (a \,\%\, b) = a$. C also defines a/b to be the result of rounding the real number a/b towards zero, so $-5/3 = -1$. Hence, $-5 \,\%\, 3 = -5 - (-5/3) * 3 = -5 + 3 = -2$ in C.

Let $\mathbf{Z}$ denote the set of all integers, positive and negative. Let $\mathbf{Z}_n \subseteq \mathbf{Z}$ contain the non-negative integers less than n, that is,

$$\mathbf{Z}_n = \{0, 1, \ldots, n-1\}.$$

We now define addition, subtraction, and multiplication operations directly on $\mathbf{Z}_n$:

$$\begin{aligned} a \oplus b &= (a + b) \bmod n \\ a \ominus b &= (a - b) \bmod n \\ a \otimes b &= (a \times b) \bmod n \end{aligned} \tag{1}$$

We will sometimes write $+, -, \times$ in place of $\oplus, \ominus, \otimes$, respectively, when it is clear from context that they are to be regarded as operations over $\mathbf{Z}_n$ rather than over $\mathbf{Z}$.

42.4 Greatest common divisor

The *greatest common divisor* of two integers a and b, written $\gcd(a, b)$, is the largest integer d such that $d \,|\, a$ and $d \,|\, b$. The $\gcd$ is always defined since 1 is a divisor of every integer, and the divisor of a number cannot be larger (in absolute value) than the number itself.

The gcd of a and b is easily found if a and b are already given in factored form. Namely, let p_i be the i^{th} prime and write $a = \prod p_i^{e_i}$ and $b = \prod p_i^{f_i}$. Then $\gcd(a, b) = \prod p_i^{\min(e_i, f_i)}$. However, factoring is believed to be a hard problem, and no polynomial-time factorization algorithm is currently known. Indeed, if it were, then Eve could use it to break RSA, and RSA would be of no interest as a cryptosystem. Fortunately, $\gcd(a, b)$ can be computed efficiently without the need to factor a and b using the famous *Euclidean algorithm*.

42.5 Euclidean algorithm

Euclid's algorithm is remarkable, not only because it was discovered a very long time ago, but also because it works without knowing the factorization of a and b. It relies on several identities satisfied by the $\gcd$ function. In the following, assume $a > 0$ and $a \geq b \geq 0$:

$$\gcd(a, b) = \gcd(b, a) \tag{2}$$

$$\gcd(a, 0) = a \tag{3}$$

$$\gcd(a, b) = \gcd(a - b, b) \tag{4}$$

Identity 2 is obvious from the definition of $\gcd$. Identity 3 follows from the fact that every positive integer divides 0. Identity 4 follows from the Fact in section 42.2.

These identities allow the problem of computing $\gcd(a, b)$ to be reduced to the problem of computing $\gcd(a - b, b)$, which is "smaller" problem as long as $b > 0$. Here we measure the size of the problem $\gcd(a, b)$ by the sum $a + b$ of the two arguments. This leads to an easy recursive algorithm shown in Figure 42.1. Nevertheless, this algorithm is not very efficient, as you will quickly discover if you attempt to use it, say, to compute $\gcd(1000000, 2)$.

Repeatedly applying (4) to the pair (a, b) until it can't be applied any more produces the sequence of pairs $(a, b), (a - b, b), (a - 2b, b), \ldots, (a - qb, b)$. The sequence stops when $a - qb < b$. But the number of times you can subtract b from a while remaining non-negative is just the quotient $\lfloor a/b \rfloor$, and the amount $a - qb$ that is left is just the remainder $a \bmod b$. Hence, one can go directly from the pair (a, b) to the pair $(a \bmod b, b)$, giving the identity

$$\gcd(a, b) = \gcd(a \bmod b, b). \tag{5}$$

```
1  int gcd(int a, int b)
2  {
3    if ( a < b ) return gcd(b, a);
4    else if ( b == 0 ) return a;
5    else return gcd(a-b, b);
6  }
```

Figure 42.1: Simple (but inefficient) $\gcd$ algorithm.

Replacing `a-b` with `a%b` in line 5 of Algorithm 42.1 (using C notation) yields an exponentially faster algorithm, one that can be shown to require at most in $O(n)$ stages, where n is the sum of the lengths of a and b when written in binary notation, and each stage involves at most one remainder computation. In addition, line 3 of Algorithm 42.1 can be eliminated if the arguments to the recursive call in line 5 are reversed. In this way, we have $a \geq b$ for all but the top-level call on $\gcd(a, b)$, eliminating the roughly half of the recursive calls whose only effect is to swap the order of arguments.

The full Euclidean algorithm is shown in Figure 42.2.

```
1  int gcd(int a, int b)
2  {
3    if ( b == 0 ) return a;
4    else return gcd(b, a%b);
5  }
```

Figure 42.2: The Euclidean algorithm.