01204211 Discrete Mathematics
Lecture 17: Primality testing (2)

Jittat Fakcharoenphol

October 31, 2015

Efficient algorithms

This lecture proves that the two forms of the Fermat's Little Theorem are equivalent.
We then outline a primality testing algorithm based on the Fermat's Little Theorem. This algorithm requires two subroutines for

- computing powers a^n,
- finding the greatest common divisors.

We will present efficient algorithms for these two problems.

Fermat's Little Theorem

The common form of the Fermat's Little Theorem is as follows.

First form

Theorem: If p is a prime and a is an integer not divisible by p, then $a^{p-1} \bmod p = 1$.

However, we prove the following version in the previous lecture:

Second form

Theorem: If p is a prime, then $p | a^p - a$, for any integer a.

We shall prove that both forms are equivalent.

Basic claim

We need the following claim to prove the equivalence.

Claim 1: If p is a prime and $p|ab$ but $p \not| a$, then $p|b$.

Proof: Factor ab into a product of primes. Since p divides ab then p must be one of the factors in the prime factorization of ab. However, since $p \not| a$, p does not appear in a's prime factorization. This implies that p must be from b's prime factorization. Hence, $p|b$. $\blacksquare$

Note that the assumption that p is a prime is crucial. For example, if we let $p = 6$, $a = 9$, and $b = 2$, we can see that $6|18$, but $6 \not| 9$ or $6 \not| 2$.

Another form for the Fermat's Little Theorem

To show that both forms of the Fermat's Little Theorem are equivalent, we need to prove the following two claims. (You should look at the claims and try to understand why proving both of them implies that both forms of the Fermat's Little Theorem are equivalent.)

Second form $\Rightarrow$ first form

Claim 2: If p is a prime, p does not divide an integer a, and $p | a^p - a$; then $a^{p-1} \bmod p = 1$.

Proof: Write $a^p - a = a(a^{p-1} - 1)$. Since $p | a(a^{p-1} - 1)$ but $p \not| a$, from claim 1, we have that $p | a^{p-1} - 1$. Thus,

$$a^{p-1} \bmod p = 1,$$

as required. ■

First form $\Rightarrow$ second form

Claim 3: Lat p be a prime. If for any integer a not divisible by p, $a^{p-1} \mod p = 1$, then for any integer b, we have that $p|b^p - b$.

Proof: There are two cases, when $p \not| b$ and $p|b$.
In the first case where $p \not| b$, the assumption implies that $p|b^{p-1} - 1$; thus, $p|b(b^{p-1} - 1)$, as required.
Now consider the case when $p|b$. Since we can write $b^p - b = b(b^{p-1} - 1)$, we know that $p|b(b^{p-1} - 1)$, because $p|b$. ■

The test: idea

If we want to check if q is a prime, from the Fermat's Little Theorem, we know that when q is a prime, for any integer a not divisible by q, $a^{q-1} \bmod q = 1$. We can devise a test based on this fact:

```
PROCEDURE FermatTest0($q$,$a$)
1. Find $y = a^{q-1} \bmod q$
2. if $y \neq 1$ then
3.   return "COMPOSITE"
4. else
5.   return "PRIME"
6. endif
```

To implement this test efficiently, we need a way to find $a^{q-1} \bmod q$ quickly. Recall that in our test, q will be a very large number and an $O(q)$-time algorithm or even $O(\sqrt{q})$-time algorithm will not be fast enough.

Computing powers

If we want to find $a^n \mod q$, a straight-forward implementation is to perform n multiplications. When a and q are m-bit numbers, this takes $O(nm^2)$ time because multiplication and division run in $O(m^2)$ time.
Can we compute the power faster?
Let's start with some example.

- How to compute a^2? We can't do anything faster then just computing directly a^2.
- What can we compute if we know a^2? Note that we can square a^2 to get $a^2 \times a^2 = a^4$.
- We can keep squaring to get $a^8 = a^4 \times a^4$, $a^{16} = a^8 \times a^8$, and so on.
- Thus, we can compute a^{2^k} with only k multiplications.

We know how to compute the power a^n when n is a power of 2.
We still need to handle the case when the exponent n is not a power of 2.

To deal with general case, we can either keep all computed powers and multiply appropriate results to get the final answer. For example, to compute a^{20}, we note that $a^{20} = a^{16} \cdot a^4$, so we first compute a^2, a^4, a^8, a^{16} and multiply a^{16} and a^4.

A recursive algorithm below takes another approach. It first finds $a^{\lfloor n/2 \rfloor}$, where $\lfloor x \rfloor$ is a floor function that returns the largest integer not greater than x (e.g., $\lfloor 1.2 \rfloor = 1$ and $\lfloor 2 \rfloor = 2$). It then squares the result and depending on the value of $n/2$, multiplies the result with a.

```
PROCEDURE Power($a$,$n$)
1. if $n = 1$ then return $a$ endif
2. let $b \leftarrow$ Power($a$,$\lfloor n/2 \rfloor$)
3. if $2|n$ then
4.   return $b \times b$
5. else
6.   return $b \times b \times a$
7. endif
```

The number of multiplications

Let's analyze the number of multiplications required to compute a^n. We will not be very precise here, and you will learn more about this techniques in later courses.

First, we can see that multiplications appear on line 4 and 6.

Excluding the multiplications excuted in the recursive calls, each invocation of procedure Power requires at most 3 multiplications.

To analyze the number of multiplications, we only need to look at the depth of the recursion.

Let's look at an example of Power(a,140):

Power(a,135) $\rightharpoonup$ Power(a,67) $\rightharpoonup$ Power(a,33) $\rightharpoonup$ Power(a,16) $\rightharpoonup$ Power(a,8) $\rightharpoonup$ Power(a,4) $\rightharpoonup$ Power(a,2) $\rightharpoonup$ Power(a,1)

One important observation is that each recursive call decreases n by at least a factor of 2.

Since for each level of recursion, n decrease by at least a factor of 2. Let n_i be the value of variable n after i levels of recursion. Clearly, $n_0 = n$. We have that $n_1 \leq n_0/2$, $n_2 \leq n_1/2$, and so on, or in general,

$$n_{i+1} \leq n_i/2.$$

This implies that

$$n_i \leq n_0/2^i = n/2^i.$$

Also, note that the algorithm stops when $n = 1$. Let k be the maximum level of recursion. We then have this equation

$$1 = n_k \leq n/2^k,$$

or $2^k \leq n$. To solve for k, we can use logarithms. Taking logarithms on both side, we get that

$$k \leq \log n,$$

where $\log(\cdot)$ is a logarithm based 2. Thus, the number of multiplications required is $O(\log n)$.

The running time

We have shown that the number of multiplications is $O(\log n)$. However, we treat all multiplications as a unit of computation. While this is OK, when the numbers are small enough. However, in our case where we deal with large numbers (e.g., 1000-bit numbers), we need to consider the size of the integers when multiplying them.

Let a and n be m-bit integers. The number of bits for a^n is nm, which is extremely large. If we need the actual number, there is no way to find it efficiently, as the number of bits in the output is already exponential in m. (Recall that n can be as large as 2^m.)

But in our case, we do not want the actual number a^n, but $a^n \mod q$, where q is an m-bit number. Therefore, we can keep all numbers below q by performing a modulo operation after every multiplication. This means that each multiplication runs in $O(m^2)$ time. Therefore, the algorithm runs in $O(m^2 \log n)$. Since n is an m-bit integer, it is at most 2^m, implying $\log n \leq m$. The running time for Power is $O(m^3)$.

The greatest common divisors

Another important concept in number theory is the greatest common divisor.
A **common divisor** for integers a and b is an integer c such that $c|a$ and $c|b$. A common divisor c is **the greatest common divisor** if, for every common divisor c', $c' \leq c$. We denote the greatest common divisor of a and b as $gcd(a, b)$.

Examples

$$gcd(100, 15) = 5.$$

$$gcd(7, 15) = 1.$$

$$gcd(1000, 120) = 40.$$

$$gcd(2558, 2530) = 2.$$

Finding the greatest common divisors

- The most straight-forward algorithm for finding the gcd of two integers a and b is to iteratively divide both numbers by integers from 1 to $\min\{a, b\}$.
- However, this will not be a polynomial time algorithm.
- We may want to use the trick from the prime testing algorithm where we only divide the numbers up to their square roots, i.e., we test for numbers from 1 to $\sqrt{\min\{a, b\}}$.
- But this does not work. *Can you find an example where this approach is broken?*

The Euclidean algorithm

The following Euclidean algorithm finds the gcd of a and b when both integers are non-negative.

The Euclidean algorithm

```
PROCEDURE GCD($a$,$b$)
1. if $a \mod b = 0$ then
2.   return $b$
3. else
4.   return GCD($b$, $a \mod b$)
5. endif
```

Note that we use GCD(a,b) to denote the output of this algorithm but we use $gcd(a, b)$ (small caps/math fonts) to denote the **correct** greatest common divisor.

An example

Suppose we want to find $gcd(12, 81)$. The execution of GCD(12,81) works as follows:

	a	b	$b\|a$?	$a \mod b$	result
GCD(15,81)	15	81	no	15	-
GCD(81,15)	81	15	no	6	-
GCD(15,6)	15,	6	no	3	-
GCD(6,3)	6	3	yes	-	3

Why does this work?

First, let's try to understand why this algorithm works.

- The first case is clear: since the largest divisor for b is b itself, when $b|a$, b must be the gcd.
- Now, suppose that $b \not| a$. Suppose that c is a common divisor. Since c divides b, it must divide any multiple of b, i.e. $c|kb$ for any integer k. Note also that we can write

  $$a \mod b = a - kb,$$

  for some integer k. Therefore, if c divides both a and b, it must also divide $a \mod b$.
- On the other hand, if an integer c divides both b and $a \mod b$. We can show that c divides a, because we can write

  $$a = kb + (a \mod b),$$

  for some integer k.

- From the previous discussion, we know that the set of common divisors of a and b, and of b and $a \bmod b$ are the same.
- Therefore, the maximum of these divisors must be the same, i.e.,
$$gcd(a, b) = gcd(b, a \bmod b).$$
- This explains why the "else" part is correct.
- To get a formal proof of the algorithm's correctness, we need mathematical induction. We leave this as an exercise.

The running time

- To prove the running time, we must be able to argue about the progress the algorithm makes.
- Before we start, let's make an assumption that $a \geq b$.
  - Note that this is always true, except in the first step. But after the first step this is always true since $a \mod b < b$.
  - Therefore, if $a < b$, we spend one iteration of GCD and then our assumption is always true.
- This also ensure that the values of variable b are decreasing as $a \mod b$ is always smaller than b. This shows that the algorithm terminates in at most b steps (*why?*), but it is not strong enough to give us a good bound on the running time.

Let's try to look at how the values of parameters a and b chan ges over time. Let a_i and b_i be the values of variables a and b at the i-th iteration. (Note that $a = a_1$ and $b = b_1$.) The following table shows a few values of a_i's and b_i's.

i	a_i	b_i
1	a	b
2	b	$a \bmod b$
3	$a \bmod b$	$b \bmod (a \bmod b)$

Note that the values of variable a change from a to b and to $a \bmod b$. If we want a polynomial running time (in the number of bits), we want a or b to decrese very quickly.

Let's try to focus on a. Can we say that a always decreases by a factor of 2 in every iteration?

- Probably not. But we know that, if $b \leq a/2$, this is true, i.e., $a_{i+1} = b_i \leq a_i/2$.
- How about the case when $b > a/2$? Can we say anything in this case?
- Clearly, in iteration $i + 1$, $a_{i+1} = b_i$, which can be larger than a_i. But let's look at the next iteration $i + 2$. Now $a_{i+2} = b_{i+1} = a_i \mod b_i$. Note that since $b_i > a_i/2$, $a_i \mod b_i = a_i - b_i < a_i/2$. Thus, we can conclude that $a_{i+2} \leq a_i/2$.

Therefore, in any case, we know that after at least two iterations, a decreases by at least a factor of 2. Clearly the algorithm stops when $a \leq 1$. Therefore, the number of iterations is at most the number of times we can divide a by 2 until a is no larger than 1. This is at most $\log a$.

If we consider the case where $b > a$, we have to account for the fact that we swap a and b. The number of iterations is $O(\max\{\log a, \log b\})$, or equivalently $O(\log a + \log b)$.
If a and b are m-bit numbers, each iteration runs in time $O(m^2)$ (the time for a modulo operation). Thus the algorithm runs in time

$$O((\log a + \log b)m^2) = O(m^3),$$

because $\log a \leq m$ and $\log b \leq m$.

The final Fermat test: idea + more condition

We can incorporate the gcd computation in our primality testing algorithm.
If q is a prime and a is chosen so that $2 \leq a \leq q-1$, we know that $gcd(q, a) = 1$. We shall add this condition to the test as well.
While this check might not be that helpful in itself when q is composite, it is crucial to our proof that the algorithm works (described in the next lecture).

```
PROCEDURE FermatTest($q$,$a$)
1. if $GCD(q, a) \neq 1$ then return "COMPOSITE" endif
2. Find $y = a^{q-1} \bmod q$
3. if $y \neq 1$ then
4.   return "COMPOSITE"
5. else
6.   return "PRIME"
7. endif
```