CMPT 307 : Algorithms with numbers (Study Guide)
Should be read in conjunction with the text
September 13, 2019

The chapter is mainly concerned with problems involving very large integers. Factoring and primality testing are two very important problems in number theory.

Suppose N is a given integer. We assume that N is very large. In binary system, N can be represented using exactly $\lceil \log_2(N+1) \rceil$ bits. In decimal system, N can be represented using exactly $\lceil \log_{10}(N+1) \rceil$ decimal digits. Note that $\log_2 N \in \Theta(\log_b N)$ for any constant base $b \geq 2$. You should know how they are exactly related. In the following, we use n as the number of bits used to represent N. An algorithm, that takes N as input (requiring n bits), runs in polynomial time if the running time is $O(n^k)$ where k is a constant integer. An $O(N)$ time algorithm is in effect an exponential time algorithm in n, the input size. As a matter of fact, any $O(N^\alpha), \alpha > 0$ time algorithm is an exponential time algorithm.

1 Arithmetic operations

Let N_1 and N_2 be two n-bit integers. The running times to compute the following operations using the grade-school arithmetic are as follows.

- $N_1 + N_2$: takes $O(n)$ time (requiring at most $n+1$ bits),
- $N_1 - N_2$: takes $O(n)$ time (requiring at most $n+1$ bits),
- $N_1 * N_2$: takes $O(n^2)$ time (requiring at most $2n$ bits), and
- N_1 / N_2: takes $O(n^2)$ time (requiring at most $2n$ bits).

Multiplying (dividing) N by 2^k (2^{-k}) takes $O(n)$ time since $2^k N$ $(2^{-k}N)$ is equivalent to shifting the n bits representing N to the left (right) by k bits. The result can be stored using $n+k$ $(n-k)$ bits. We have assumed $k \leq n$.

Note that both multiplication and division can be computed recursively. These can be described as follows:

- **Multiplication** Compute $x.y$, where $y \geq 0$.

$$x.y = \begin{cases} 2(x.\lfloor \frac{y}{2} \rfloor) & \text{if y is even} \\ x + 2(x.\lfloor \frac{y}{2} \rfloor) & \text{if y is odd} \end{cases}$$

Apply the above algorithm on the input: $x = 17$ and $y = 21$. Show the various stages of the recursive algorithm described in Figure 1.1 of the text.

- **Division** When x is divided by y, $y \geq 1$, we get $x = qy + r$ where q is the quotient and $r < y$ is the remainder. The division algorithm is described in Fig. 1.2 in the text. Try to determine, like multiplication, the recursive definition of division.

1.1 Evaluating x^N: x and N are m and n bit integers, respectively

If we compute $x^N = x \times x \times \ldots \times x$ (N times), the number of multiplications needed is $N-1$. This approach leads to an exponential time algorithm. The number of bits required to store the result is Nm, which is exponential. We can reduce the number of multiplications from N to n as follows.

Let N be $(b_n b_{n-1} \ldots b_2 b_1 b_0)_2$ in the binary system. Now we can write x^N as:

$$x^N = \prod_{i=0}^{i=n} b_i \times x^{2^i}.$$

At most $2n$ multiplications are needed to compute x^N (why?).

The above algorithm is the same as the following recursive algorithm (why?).

$$x^N = \begin{cases} (x^{\lfloor \frac{N}{2} \rfloor})^2 & \text{if N is even} \\ x(x^{\lfloor \frac{N}{2} \rfloor})^2 & \text{if N is odd} \end{cases}$$

Note that the space requirement to store the result is still exponential in n. In the following we show that modular arithmetic is an arithmetic system that deals with restricted ranges of integers.

2 Modular arithmetic

x modulo N (or **x** mod **N**) is the remainder when x is divided by N. When two integers x and y are such that $x \bmod N = y \bmod N$, we say x and y are **congruent modulo** N and we write $x \equiv y (\bmod N)$. Thus

Lemma For a positive integer x and a natural number N, if $r = a \bmod N$, then

- $a \equiv r (\bmod N)$;

- $r \in \{0, 1, 2, \ldots, N-1\}$.

mod N relation divides the set of integers $\mathbb{Z}$ into N equivalent classes. When $N = 3$, the classes are:

$$[0] = \{i | i \equiv 0 (\bmod N)\} = \{\ldots, -6, -3, 0, 3, 6, \ldots\}$$

$$[1] = \{i | i \equiv 1 (\bmod N)\} = \{\ldots, -5, -2, 0, 1, 4, \ldots\}$$

$$[2] = \{i | i \equiv 2 (\bmod N)\} = \{\ldots, -4, -1, 0, 2, 5, \ldots\}$$

2.1 Substitution rule

The following two rules define how we perform modular addition and multiplication. Suppose $x \equiv x' (\bmod N)$ and $y \equiv y' (\bmod N)$.

Modular addition: $x + y \equiv x' + y' (\bmod N)$

Modular multiplication: $xy \equiv x'y' (\bmod N)$

Modular addition and multiplication respectively cost $O(n)$ and $O(n^2)$ as well.

Suppose we know $a \equiv b (\bmod N)$. Then the modular multiplication rule implies that

$a^2 \equiv b^2 (\bmod N)$,

$a^3 \equiv b^3 (\bmod N)$,

...

...

$a^k \equiv b^k (\bmod N)$, for positive k

2.2 Modular exponentiation

We are interested in computing $x^y \bmod N$ where x, y, N are all n-bit integers.

This is very similar to evaluating x^y. The recursive algorithm we can use is

$$x^y (\bmod N) \equiv \begin{cases} (x^{\lfloor \frac{y}{2} \rfloor})^2 \ (\bmod N) & \text{if y is even} \\ x(x^{\lfloor \frac{y}{2} \rfloor})^2 (\bmod N) & \text{if y is odd} \end{cases}$$

The pseudocode of the above algorithm is given in Figure 1.4 of the text. The space complexity in this case is O(n) (note that it is exponential when evaluating just x^y). Thus modular exponentiation can be done in cubic time in n (why?). The space requirement is linear in n. This is one of the great advantages of modular arithmetic.

Example: Reduce $d = 1829764 \mod 11$.

Ans: we can write

$$\begin{aligned} 1829764 &= 4.10^0 + 6.10^1 + 7.10^2 + 9.10^3 + 2.10^4 + 8.10^5 + 1.10^6 \\ &\equiv 4.1 + 6.10 + 7.1 + 9.10 + 2.1 + 8.10 + 1.1 \pmod{11} \\ &\equiv 4 + 60 + 7 + 90 + 2 + 80 + 1 \pmod{11} \\ &\equiv 244 \pmod{11} \\ &\equiv 2 \pmod{11} \end{aligned}$$

Using simple division by 11 for the reduction will result in an exponential time algorithm. The above algorithm is $O((\log N)^2)$.

2.3 Greatest Common Divisor

Given two integers a and b, determine the largest integer that divides both of them. The largest integer is know as the greatest common divisor (gcd). **gcd(14,35) is 7**. Two integers a and b are called relatively prime if gcd(a,b)=1. The gcd algorithm is guided by the following rule.

Euclid Rule: If x and y are positive integers with $x \geq y$, then $gcd(x,y) = gcd(y, x \bmod y)$.

The algorithm presented in Fig. 1.5 of the text follows immediately from the definition. The running time of the algorithm is dependent on the number of times the routine Euclid is called. For the analysis purpose, let us rewrite the algorithm in Fig. 1.5 as:

```
function Euclid-modified(a,b)
Input: two integers a, b with a >= b >=0
Output: gcd(a,b)

if b = 0 return a
r1 = a mod b
if r1 = 0 return b
r2 = b mod r1
```

```
return Euclid-modified(r1, r2)
```

The lemma in page 21 of the text shows that $r_1 < \frac{a}{2}$ and $r_2 < \frac{b}{2}$. This implies that Euclid-modified will be called at most $O(n)$ times. Each call of Euclid-modified requires the evaluation of a mod function which costs $O(n^2)$ running time. Therefore, the time complexity of Euclid(a,b) or Euclid-modified(a,b) is $O(n^3)$.

2.3.1 Computing gcd(252, 198)

If we apply Euclid(252, 198) we get the following.

1. $252 = 1 \times 198 + 54$
2. $198 = 3 \times 54 + 36$
3. $54 = 1 \times 36 + 18$
4. $36 = 2 \times 18 + 0$

Thus 18 is the greatest common divisor of 252 and 198.

Next we show that 18, the gcd of 252 and 198, can be expressed in terms of 252 and 198.

1. $18 = 54 - 1 \times 36$ (from step 3)
2. $18 = 54 - (198 - 3 \times 54) = 4 \times 54 - 1 \times 198$ (from step 2)
3. $18 = 4 \times (252 - 1 \times 198) - 1 \times 198 = 4 \times 252 - 5 \times 198$ (from step 1)

Thus $18 = 4 \times 252 - 5 \times 198$. As a matter of fact one can show that

Lemma: (page 22 of the text) There exist integers x and y such that $gcd(a,b) = ax + by$.

The converse is also true.

Lemma: (page 21 of the text) If d divides both a and b, and $d = ax + by$ for some integers x and y, then necessarily $d = gcd(a,b)$.

2.3.2 How do you compute $gcd(a,b) = ax + by$?

The recursive algorithm in Fig. 1.6 in the text computes it. It is based on the following observations.

The first two steps of the computation of $gcd(a,b)$ are

1. $a = tb + (a \bmod b)$ where $t = \lfloor \frac{a}{b} \rfloor$ is the quotient
2. $gcd(b, a \bmod b)$

Suppose we recursively compute $gcd(b, a \bmod b) = x'b + y'(a \bmod b)$, which is $gcd(a,b) = x'b + y'(a \bmod b)$. Now $a \bmod b = a - tb$ from step 1. Therefore, $gcd(a,b) = x'b + y'(a - tb) = y'a + (x' - ty')b = y'a + (x' - \lfloor \frac{a}{b} \rfloor y')b$. Thus we can compute x, y such that $gcd(a,b) = ax + by$ in $O(n^3)$ time (why?).

2.4 Modular division

We say that x is the multiplicative inverse of $a \bmod N$ if $ax \equiv 1 (\bmod N)$. Note that for even N, when $a = 2$, there does not exist any x such that $2x \equiv 1 (\bmod N)$.

Suppose $gcd(a,N) = 1$, i.e a and N are relatively prime. We know from the previous section that there exists integers x and y such that 1= ax + Ny. Therefore, $ax = 1 (\bmod N)$.

The above observations give us an algorithm to compute the multiplicative inverse of $a \bmod N$. First compute $gcd(a,N)$. If it is not equal to 1, the multiplicative inverse does not exist. Otherwise, compute $1 = ax + Ny$ (it is always possible). Then return x as the multiplicative inverse. This can be done in $O(n^3)$ time (why?).

3 Primality testing

Fermat's little theorem is the most important theorem for the Primality testing problem. It says

Fremat's little theorem: If p is a prime, for every $a, 1 \leq a < p$, $a^{p-1} \equiv 1 (\bmod p)$.
Proof: Consider the sequence $a \times 1 (\bmod p), a \times 2 (\bmod p), \ldots, a \times (p-1) (\bmod p)$ for any $1 \leq a < p$. We can show that (look at the text) none of these is zero, and no pair is equal. Therefore,

$$\prod_{i=1}^{p-1} a \times i \equiv \prod_{i=1}^{p-1} i (\bmod p).$$

The above simplifies to

$$a^{p-1} \equiv 1 (\bmod p).$$

Fermat's little theorem is important in the design of primality testing algorithms (without factoring).

We are interested in solving the problem:

Given an integer N, efficiently decide if it is a prime.

We are interested in an algorithm that takes $O(\log^{O(1)} N)$(i.e. $O((\log N)^k, k$ constant$)$) time.

3.1 School Method

The primality of N is tested by dividing all numbers $< N$ or better, $\leq \sqrt{N}$. This method requires $\sqrt{N}$ divisions, i.e. $2^{\frac{1}{2}\log_2 N}$ divisions which is exponential in $\log_2 N$. We need a better algorithm.

3.2 A simple algorithm based in Fermat's Little Theorem (FLT)

The algorithm for primality testing is:

1. Randomly select m different a's (set M), then test if $a^{N-1} \equiv 1 (\bmod N)$.
2. If any one of the tests does not satisfy $a^{N-1} \equiv 1 (\bmod N)$, return **N is composite**.
3. Otherwise, return **N is prime**.

The running time of the algorithm is $O(m.n^2)$ which is quadratic in n when m is a constant.

Clearly, the above algorithm is correct when the output message is " N is composite ". If the output message is " N is prime", N need not be prime. This can happen in one of the following two ways.

3.2.1 There exists an integer $t < N$ where $t^{N-1} \not\equiv 1 (\bmod N)$ and M does not contain any such t.

It is shown in the text that if N is not a prime, then $t^{N-1} \equiv 1 (\bmod N)$ can happen for at most half of the values of $t < N$. Since a's are selected randomly, the probability of M not containing a single element t such that $t^{N-1} \not\equiv 1 (\bmod N)$

is $\frac{1}{2^k}$, where k is the size of M. We can therefore reduce this one-sided error by applying FLT many times, by randomly picking several values of $a < N$ and testing them all. Thus

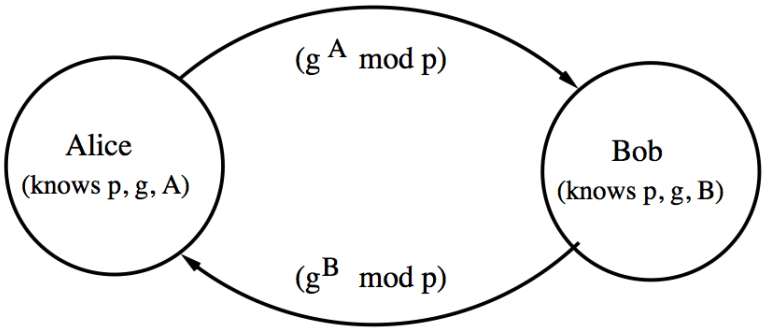
$$\Pr(\text{above algorithm returns yes when N is not prime}) \leq \frac{1}{2^k}.$$

3.2.2 There are composite numbers with the property that for every $a < N$, $a^{N-1} \equiv 1 (\bmod n)$

These numbers are called Carmichael numbers. $561 = 3 \times 11 \times 17$ is a Carmichael number. On Carmichael numbers, the primality test algorithm described above will fail. These numbers are pathologically rare. These numbers can be taken care of separately. We will not deal with them. We can claim that in a Carmichael-free universe, with probability $\frac{1}{2^k}$, the algorithm returns the correct answer.

4 Diffie-Hellman (D-H) Algorithm

The D-H key agreement protocol (1976) was the first practical method for establishing a shared secret over an unsecured channel. The current RSA protocol is slightly different. The protocol can be visualized as follows:



The steps of the D-H protocol between Alice and Bob are as follows.

1. Alice and Bob agreed on two public keys: a prime number p and a base g.
2. Alice chooses a secret number A, and sends Bob ($g^A \bmod p$).

3. Bob chooses a secret number B, and sends Alice ($g^B \bmod p$).

4. Alice computes $((g^B \bmod p)^A mod p)$.

5. Bob computes $((g^A \bmod p)^B mod p)$.

6. Both Alice and Bob have the same key which they can use to encrypt and decrypt the data. Not that p and g are public keys. The secret keys were never transmitted.

An example

- Alice and Bob agrees on $p = 23$ and $g = 5$.
- Alice chooses his private key $A = 6$ and sends Bob $5^6 \bmod 23 = 8$.
- Bob chooses his private key $B = 15$ and sends Alice $5^{15} \bmod 23 = 19$.
- Alice computes $19^6 \bmod 23 = 2$.
- Bob computes $8^6 \bmod 23 = 2$.

Note that $(g^B \bmod p)^A mod p = ((g^{BA} \bmod p) \bmod p)$. Similarly, $(g^A \bmod p)^B \bmod p = ((g^{AB} \bmod p) \bmod p)$.

Consider the difficulty of deciphering when A, B, p, g are very large (requiring hundreds of bits).