Algorithms (II)

Yu Yu

Shanghai Jiaotong University

Chapter 1. Algorithms with Numbers

Two seemingly similar problems

Factoring: Given a number N, express it as a product of its prime factors.

Primality: Given a number N, determine whether it is a prime.

We believe that **Factoring** is hard and much of the electronic commerce is built on this assumption.

There are efficient algorithms for **Primality**, e.g., AKS test by **Manindra Agrawal**, **Neeraj Kayal**, and **Nitin Saxena**.

Basic arithmetic

How to represent numbers

We are most familiar with decimal representation:

$$1024.$$

But computers use binary representation:

$$1\underbrace{0\ldots0}_{\text{10 times}}.$$

The bigger the base is, the shorter the representation is. But how much do we really gain by choosing large base?

Bases and logs

Question

How many digits are needed to represent the number $N \geq 0$ in base b?

Answer:

$$\lceil \log_b(N+1) \rceil$$

Question

How much does the size of a number change when we change bases?

Answer:

$$\log_b N = \frac{\log_a N}{\log_a b}.$$

In big-O notation, therefore, the base is irrelevant, and we write the size simply as $O(\log N)$.

The roles of $\log N$

1. $\log N$ is the power to which you need to *raise 2* in order to obtain N.
2. Going backward, it can also be seen as the number of times you must *halve N* to get down to 1. (More precisely: $\lceil \log N \rceil$.)
3. It is the number of bits in the binary representation of N. (More precisely: $\lceil \log(N+1) \rceil$.)
4. It is also the depth of a complete binary tree with N nodes. (More precisely: $\lfloor \log N \rfloor$.)
5. It is even the sum $1 + 1/2 + 1/3 + \ldots + 1/N$, to within a constant factor.

Addition

The sum of any three single-digit numbers is at most two digits long.

In fact, this rule holds not just in decimal but in any base $b \geq 2$.

In binary, for instance, the maximum possible sum of three single-bit numbers is 3, which is a 2-bit number.

This simple rule gives us a way to add two numbers in any base: align their right-hand ends, and then perform a single right-to-left pass in which the sum is computed digit by digit, maintaining the overflow as a carry. Since we know each individual sum is a two-digit number, the carry is always a single digit, and so at any given step, three single-digit numbers are added.

Addition (cont'd)

Carry:	1			1	1	1		
		1	1	0	1	0	1	(53)
		1	0	0	0	1	1	(35)
	1	0	1	1	0	0	0	(88)

Ordinarily we would spell out the algorithm in pseudocode, but in this case it is so familiar that we do not repeat it.

Addition (cont'd)

Question

Given two binary numbers x and y, how long does our algorithm take to add them?

We want the answer expressed as a **function of the size of the input**: the number of bits of x and y.

Suppose x and y are each n bits long. Then the sum of x and y is $n+1$ bits at most, and each individual bit of this sum gets computed in a fixed amount of time.
The total running time for the addition algorithm is therefore of the form $c_0 + c_1 n$, where c_0 and c_1 are some constants, i.e., $O(n)$.

Addition (cont'd)

Question

Is there a faster algorithm?

In order to add two n-bit numbers we must at least read them and write down the answer, and even that requires n operations.
So the addition algorithm is *optimal*, up to multiplicative constants!

Does the usual programs perform addition in one step?

1. A single instruction we can add integers whose size in bits is within the word length of today's computer – 64 perhaps. But it is often useful and necessary to handle numbers much larger than this, perhaps several thousand bits long.
2. When we want to understand algorithms, it makes sense to study even the basic algorithms that are encoded in the hardware of today's computers. In doing so, we shall focus on the bit complexity of the algorithm, the number of elementary operations on individual bits, because this accounting reflects the amount of hardware, transistors and wires, necessary for implementing the algorithm.

Multiplication

				1	1	0	1	(binary 13)
			$\times$	1	0	1	1	(binary 11)
				1	1	0	1	(1101 times 1)
			1	1	0	1		(1101 times 1, shifted once)
		0	0	0	0			(1101 times 0, shifted twice)
$+$	1	1	0	1				(1101 times 1, shifted thrice)
1	0	0	0	1	1	1	1	(binary 143)

Multiplication (cont'd)

The grade-school algorithm for multiplying two numbers x and y is to create an array of *intermediate sums*, each representing the product of x by a single digit of y. These values are appropriately left-shifted and then added up.

If x and y are both n bits, then there are n intermediate rows, with lengths of up to $2n$ bits (taking the shifting into account). The total time taken to add up these rows, doing two numbers at a time, is

$$\underbrace{O(n) + \ldots + O(n)}_{n-1 \text{ times}}.$$

which is $O(n^2)$.

Al Khwarizmi's algorithm

To multiply two decimal numbers x and y, write them next to each other. Then repeat the following:

> *divide the first number by* 2, *rounding down the result (that is, dropping the .5 if the number was odd), and double the second number.*

Keep going till the first number gets down to 1. Then *strike out all the rows in which the first number is even*, and add up whatever remains in the second column.

Multiplication a la Français

MULTIPLY(x, y)
// Two n-bit integers x and y, where $y \geq 0$.

1. **if** $y = 0$ **then** return 0
2. $z =$ MULTIPLY$(x, \lfloor y/2 \rfloor)$
3. **if** y is even **then** return $2z$
4. **else** return $x + 2z$

Another formulation:

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

Multiplication a la Français (cont'd)

Question

How long does the algorithm take?

Answer: It must terminate after n recursive calls, because at each call y is halved. And each recursive call requires these operations: a division by 2 (right shift); a test for odd/even (looking up the last bit); a multiplication by 2 (left shift); and possibly one addition, a total of $O(n)$ bit operations. The total time taken is thus $O(n^2)$.

Question

Can we do better?

Answer: Yes.

Division

DIVIDE(x, y)
// Two n-bit integers x and y, where $y \geq 1$.

1. **if** $x = 0$ **then** return $(q, r) = (0, 0)$
2. $(q, r) = \text{DIVIDE}(\lfloor x/2 \rfloor, y)$
3. $q = 2 \cdot q$, $r = 2 \cdot r$
4. **if** x is odd **then** $r = r + 1$
5. **if** $r \geq y$ **then** $r = r - y$, $q = q + 1$
6. return (q, r)

Modular arithmetic

Modular arithmetic is a system for dealing with restricted ranges of integers.

We define x modulo N to be the remainder when x is divided by N; that is, if $x = qN + r$ with $0 \leq r < N$, then x modulo N is equal to r.

x and y are congruent modulo N if they differ by a multiple of N, i.e.,

$$x \equiv y \pmod{N} \quad \Longleftrightarrow \quad N \text{ divides } (x - y).$$

Two interpretations

1. It limits numbers to a predefined range $\{0, 1, \ldots, N\}$ and wraps around whenever you try to leave this range – like the hand of a clock.
2. Modular arithmetic deals with all the integers, but divides them into N equivalence classes, each of the form $\{i + k \cdot N \mid k \in \mathbb{Z}\}$ for some i between 0 and $N - 1$.

Two's complement

Modular arithmetic is nicely illustrated in two's complement, the most common format for storing signed integers.

It uses n bits to represent numbers in the range

$$[-2^{n-1}, 2^{n-1} - 1]$$

and is usually described as follows:

- Positive integers, in the range 0 to $2^{n-1} - 1$, are stored in regular binary and have a leading bit of 0.
- Negative integers $-x$, with $1 \leq x \leq 2^{n-1}$, are stored by first constructing x in binary, then flipping all the bits, and finally adding 1. The leading bit in this case is 1.

Rules

Substitution rule: If $x \equiv x' \pmod{N}$ and $y \equiv y' \pmod{N}$, then:

$$x + y \equiv x' + y' \pmod{N} \quad \text{and} \quad xy \equiv x'y' \pmod{N}$$

Algebraic rules:

$$x + (y + z) \equiv (x + y) + z \pmod{N} \qquad \text{Associativity}$$

$$xy \equiv yx \pmod{N} \qquad \text{Commutativity}$$

$$x(y + z) \equiv xy + xz \pmod{N} \qquad \text{Distributivity}$$

$$2^{345} \equiv (2^5)^{69} \equiv 32^{69} \equiv 1^{69} \equiv 1 \pmod{31}$$

Modular addition

To add two numbers x and y modulo N, we start with regular addition. Since x and y are each in the range 0 to $N-1$, their sum is between 0 and $2(N-1)$.
If the sum exceeds $N-1$, we merely need to subtract off N to bring it back into the required range.
The overall computation therefore consists of an addition, and possibly a subtraction, of numbers that never exceed $2N$.

Its running time is linear in the sizes of these numbers, in other words $O(n)$, where $n = \lceil \log N \rceil$.

Modular multiplication

To multiply two mod-N numbers x and y, we again just start with regular multiplication and then reduce the answer modulo N. The product can be as large as $(N-1)^2$, but this is still at most $2n$ bits long since

$$\log(N-1)^2 = 2\log(N-1) \leq 2n.$$

To reduce the answer modulo N, we compute the remainder upon dividing it by N, using our quadratic-time division algorithm.

Multiplication thus remains a *quadratic* operation.

Modular division

Not quite so easy.

In ordinary arithmetic there is just one tricky case – division by zero. It turns out that in modular arithmetic there are potentially other such cases as well, which we will characterize toward the end of this section.

Whenever division is legal, however, it can be managed in cubic time, $O(n^3)$.

Modular exponentiation

In the **cryptosystem** we are working toward, it is necessary to compute $x^y \mod N$ for values of x, y, and N that are *several hundred bits long*.

The result is some number modulo N and is therefore itself a few hundred bits long. However, the raw value of x^y could be much, much longer than this. Even when x and y are just 20-bit numbers, x^y is at least

$$(2^{19})^{(2^{19})} = 2^{(19)(524288)},$$

about 10 million bits long!

Modular exponentiation (cont'd)

To make sure the numbers we are dealing with never grow too large, we need to *perform all intermediate computations modulo N*.

First idea: calculate $xy \mod N$ by repeatedly multiplying by x modulo N. The resulting sequence of intermediate products,

$$x \mod N \to x^2 \mod N \to x^3 \mod N \to \cdots \to x^y \mod N$$

consists of numbers that are smaller than N, and so the individual multiplications do not take too long. But imagine if y is 500 bits long ...

Modular exponentiation (cont'd)

Second idea: starting with x and *squaring repeatedly modulo N*, we get

$$x \mod N \to x^2 \mod N \to x^4 \mod N \to x^8 \to \cdots \to x^{2^{\lfloor \log y \rfloor}} \mod N.$$

Each takes just $O(\log^2 N)$ time to compute, and in this case there are only $\log y$ multiplications.

To determine $x^y \mod N$, we simply multiply together an appropriate subset of these powers, those corresponding to 1's in the binary representation of y. For instance,

$$x^{25} = x^{11001_2} = x^{10000_2} \cdot x^{1000_2} \cdot x^{1_2} = x^{16} \cdot x^8 \cdot x^1.$$

Modular exponentiation (cont'd)

```
MODEXP(x, y, N)
// Two n-bit integers x and N, and an integer exponent y
1. if y = 0 then return 1
2. z = MODEXP(x, ⌊y/2⌋, N)
3. if y is even then return z^2 mod N
4.         else return x · z^2 mod N
```

Another formulation:

$$x^y = \begin{cases} \left(x^{\lfloor y/2 \rfloor}\right)^2 & \text{if } y \text{ is even} \\ x \cdot \left(x^{\lfloor y/2 \rfloor}\right)^2 & \text{if } y \text{ is odd.} \end{cases}$$

The algorithm will halt after at most n recursive calls, and during each call it multiplies n-bit numbers (doing computation modulo N saves us here), for a total running time of $O(n^3)$.

Euclid's algorithm for greatest common divisor

Question

Given two integers a and b, how to find their greatest common divisor ($\gcd(a, b)$)?

Euclid's rule: If x and y are positive integers with $x \geq y$, then $\gcd(x, y) = gcd(x \mod y, y)$.

Proof.

It is enough to show the slightly simpler rule

$$\gcd(x, y) = \gcd(x - y, y).$$

Any integer that divides both x and y must also divide $x - y$, so $\gcd(x, y) \leq \gcd(x - y, y)$. Likewise, any integer that divides both $x - y$ and y must also divide both x and y, so $\gcd(x, y) \geq \gcd(x - y, y)$. □

Euclid's algorithm for greatest common divisor (cont'd)

EUCLID(a, b)
// Input: two integers a and b with $a \geq b \geq 0$
// Output: gcd(a, b)

1. **if** $b = 0$ **then** return a
2. return EUCLID$(b, a \mod b)$

Lemma
If $a \geq b \geq 0$, then $a \mod b < a/2$.

Proof.
If $b \leq a/2$, then we have $a \mod b < b \leq a/2$; and if $b > a/2$, then $a \mod b = a - b < a/2$. □

Euclid's algorithm for greatest common divisor (cont'd)

EUCLID(a, b)
// Input: two integers a and b with $a \geq b \geq 0$
// Output: gcd(a, b)

1. **if** $b = 0$ **then** return a
2. return EUCLID$(b, a \mod b)$

Lemma

If $a \geq b \geq 0$, *then* $a \mod b < a/2$.

This means that after any *two consecutive rounds*, both arguments, a and b, are at the very least halved in value, i.e., the length of each decreases by at least one bit.
If they are initially n-bit integers, then the base case will be reached within $2n$ recursive calls. And since each call involves a quadratic-time division, the total time is $O(n^3)$.

An extension of Euclid's algorithm

Question

Suppose someone claims that d is the greatest common divisor of a and b: how can we check this?
It is not enough to verify that d divides both a and b, because this only shows d to be a common factor, not necessarily the largest one.

Lemma

If d divides both a and b, and $d = ax + by$ for some integers x and y, then necessarily $d = \gcd(a, b)$.

Proof.

By the first two conditions, d is a common divisor of a and b, hence $d \leq \gcd(a, b)$. On the other hand, since $\gcd(a, b)$ is a common divisor of a and b, it must also divide $ax + by = d$, which implies $\gcd(a, b) \leq d$. □

An extension of Euclid's algorithm (cont'd)

EXTENDED-EUCLID(a, b)
// Input: two integers a and b with $a \geq b \geq 0$
// Output: integers x, y, d such that $d = \gcd(a, b)$ and $ax+by = d$

1. **if** $b = 0$ **then** return $(1, 0, a)$
2. $(x', y', d) =$ EXTENDED-EUCLID$(b, a \mod b)$
3. return $(y', x' - \lfloor a/b \rfloor y', d)$

Lemma

For any positive integers a and b, the extended Euclid algorithm returns integers x, y, and d such that $\gcd(a, b) = d = ax + by$.

Proof of the correctness

$d = \gcd(a, b)$ is by the original Euclid's algorithm.

The rest is by induction on b. The case for $b = 0$ is trivial.
Assume $b > 0$, then the algorithm finds $\gcd(a, b)$ by calling $\gcd(b, a \mod b)$.
Since $a \mod b < b$, we can apply the induction hypothesis on this call and conclude

$$\gcd(b, a \mod b) = bx' + (a \mod b)y'.$$

Writing $(a \mod b)$ as $(a - \lfloor a/b \rfloor\, b)$, we find

$$\begin{aligned} d = \gcd(a, b) &= \gcd(b, a \mod b) = bx' + (a \mod b)y' \\ &= bx' + (a - \lfloor a/b \rfloor\, b)y' = ay' + b(x' - \lfloor a/b \rfloor\, y'). \end{aligned}$$

Modular inverse

We say x is the multiplicative inverse of a modulo N if $ax \equiv 1 \mod N$.

Lemma

There can be at most one such x modulo N with $ax \equiv 1 \mod N$, denoted by a^{-1}.

Remark

However, this inverse does not always exist! For instance, 2 is not invertible modulo 6.

Modular division

Modular division theorem For any $a \mod N$, a has a multiplicative inverse modulo N if and only if it is relatively prime to N (i.e., $\gcd(a, N) = 1$). When this inverse exists, it can be found in time $O(n^3)$ by running the extended Euclid algorithm.

Example

We wish to compute

$$11^{-1} \mod 25.$$

Using the extended Euclid algorithm, we find $15 \cdot 25 - 34 \cdot 11 = 1$, thus $-34 \cdot 11 \equiv 1 \mod 25$ and $-34 \equiv 16 \mod 25$.

This resolves the issue of modular division: when working modulo N, we can divide by numbers relatively prime to N. And to actually carry out the division, we multiply by the inverse.