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:

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

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 ≥ 0.
1. if y = 0 then return 0
2. z = MULTIPLY(x, ⌊y/2⌋)
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 ≥ 1.
1. if x = 0 then return (q, r) = (0, 0)
2. (q, r) = DIVIDE(⌊x/2⌋, y)
3. q = 2 · q, r = 2 · r
4. if x is odd then r = r + 1
5. if r ≥ y then r = r − y, q = q + 1
6. return (q, r)
```

Basic arithmetic

operation	time	optimality
addition	$O(n)$	yes
multiplication	$O(n^2)$	no
division	$O(n^2)$	I don't know

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 $0 \le r < N$, 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.,

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

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 $1 \le x \le 2^{n-1}$, 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:

$x + (y + z) \equiv (x + y) + z \pmod{N}$	Associativity
$xy \equiv yx \pmod{N}$	Commutativity
$x(y + z) \equiv xy + xz \pmod{N}$	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.

operation	time
modular addition	$O(n)$
modular multiplication	$O(n^2)$

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 ≥ b ≥ 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 ≥ b ≥ 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 ≥ b ≥ 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' − ⌊a/b⌋ 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 <u>multiplicative inverse</u> 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 $\underline{a^{-1}}$.

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.

Primality Testing

Fermat's little theorem

If p is prime, then for every $1 \leq a < p$,

$$a^{p-1} \equiv 1 \pmod{p}.$$

Proof of the Fermat's little theorem:

Let $S = \{1, 2, \ldots, p-1\}$. We claim that

the effect of multiplying these numbers by a (modulo p) is simply to permute them.

Assume $a \cdot i \equiv a \cdot j \pmod{p}$. Dividing both sides by a gives

$$i \equiv j \pmod{p}.$$

They are nonzero because $a \cdot i \equiv 0 \pmod{p}$ similarly implies $i \equiv 0 \pmod{p}$. (And we can divide by a, because by assumption it is nonzero and therefore relatively prime to p.)

Proof of the Fermat's little theorem (cont'd)

We now have two ways to write set S:

$$S = \{1, 2, \ldots, p-1\} = \{a \cdot 1 \mod p, a \cdot 2 \mod p, \ldots, a \cdot (p-1) \mod p\}.$$

We can multiply together its elements in each of these representations to get

$$(p-1)! \equiv a^{p-1} \cdot (p-1)! \pmod{p}.$$

Dividing by $(p-1)!$ (which we can do because it is relatively prime to p, since p is assumed prime) then gives the theorem. $\square$

A (problematic) algorithm for testing primality

```
PRIMALITY(N)
// Input: positive integer N
// Output: yes/no
1. Pick a positive integer a < N at random
2. if a^{N-1} ≡ 1 (mod N)
3.        then return yes
4.        else return no.
```

The problem is that Fermat's theorem is not an if-and-only-if condition, e.g.,

$$341 = 11 \cdot 31, \qquad \text{and} \quad 2^{340} \equiv 1 \mod 341.$$

Our best hope: for composite N, *most values* of a will fail the test, which motivates the above algorithm: rather than fixing an arbitrary value of a in advance, we should choose it randomly from $\{1, \ldots, N-1\}$.

Carmichael numbers

Theorem

There are composite numbers N such that for every $a < N$ relatively prime to N,

$$a^{N-1} \equiv 1 \pmod{N}.$$

Example: $561 = 3 \cdot 11 \cdot 17$.

Non-Carmichael numbers

Lemma

If $a^{N-1} \not\equiv 1 \pmod{N}$ for some a relatively prime to N, then it must hold for at least half the choices of $a < N$.

Proof.

Fix some value of a for which $a^{N-1} \not\equiv 1 \pmod{N}$. Assume some $b < N$ satisfies $b^{N-1} \equiv 1 \pmod{N}$, then

$$(a \cdot b)^{N-1} \equiv a^{N-1} \cdot b^{N-1} \equiv a^{N-1} \not\equiv 1 \pmod{N}.$$

For $b \not\equiv b' \pmod{N}$ we have

$$a \cdot b \not\equiv a \cdot b' \pmod{N}.$$

The one-to-one function $b \mapsto a \cdot b \mod N$ shows that at least as many elements fail the test as pass it. □

Primality testing without Carmichael numbers

We are ignoring Carmichael numbers, so we can now assert:

If N is prime, then $a^{N-1} \equiv 1 \mod N$ for all $a < N$.

If N is not prime, then $a^{N-1} \equiv 1 \mod N$ for *at most half* the values of $a < N$.

Therefore (for non-Carmichael numbers)

$$\Pr(\text{PRIMALITY returns yes when } N \text{ is prime}) = 1$$

$$\Pr(\text{PRIMALITY returns yes when } N \text{ is not prime}) \le \frac{1}{2}.$$

An algorithm for testing primality with low error probability

```
PRIMALITY2(N)
// Input: positive integer N
// Output: yes/no
1. Pick positive integers a_1, a_2, ..., a_k < N at random
2. if a_i^{N-1} ≡ 1 (mod N) for all i = 1, 2, ..., k
3.        then return yes
4.        else return no.
```

$$\Pr(\text{PRIMALITY2 returns yes when } N \text{ is prime}) = 1$$

$$\Pr(\text{PRIMALITY2 returns yes when } N \text{ is not prime}) \le \frac{1}{2^k}.$$

Generating random primes

Lagrange's prime number theorem. Let $\pi(x)$ be the number of primes $\le x$. Then $\pi(x) \approx x/(\ln x)$, or more precisely

$$\lim_{x \to \infty} \frac{\pi(x)}{(x/\ln x)} = 1.$$

Such abundance makes it simple to generate a random n-bit prime:

1. Pick a random n-bit number N.
2. Run a primality test on N.
3. If it passes the test, output N; else repeat the process.

Generating random primes (cont'd)

Question

How fast is this algorithm?

If the randomly chosen N is truly prime, which happens with probability at least $1/n$, then it will certainly pass the test. So on each iteration, this procedure has at least a $1/n$ chance of halting.

Therefore on average it will halt within $O(n)$ rounds.

Cryptography

The typical setting for cryptography

- **Alice** and **Bob**, who wish to communicate in private.
- **Eve**, an eavesdropper, will go to great lengths to find out what Alice and Bob are saying.

Alice wants to send a *specific message x*, written in binary, to her friend Bob.

1. Alice encodes it as $e(x)$, sends it over.
2. Bob applies his *decryption function $d(\cdot)$* to decode it: $d(e(x)) = x$.
3. Eve, will intercept $e(x)$: for instance, she might be a sniffer on the network.

Ideally the encryption function $e(\cdot)$ is so chosen that without knowing $d(\cdot)$, Eve cannot do anything with the information she has picked up.

In other words, knowing $e(x)$ tells her little or nothing about what x might be.

An encryption function:

$$e : \langle \text{messages} \rangle \to \langle \text{encoded messages} \rangle.$$

e must be *invertible* – for decoding to be possible – and is therefore a *bijection*. Its inverse is the decryption function $d(\cdot)$.

Private-key schemes: one-time pad

- Alice and Bob meet beforehand and secretly choose a binary string r of the same length–say, n bits–as the important message x that Alice will later send.
- Alice's encryption function is then a **bitwise exclusive-or**,

$$e_r(x) = x \oplus r.$$

- This function e_r is a bijection from n-bit strings to n-bit strings, as it is its own inverse:

$$e_r(e_r(x)) = (x \oplus r) \oplus r = x \oplus (r \oplus r) = x \oplus \bar{0} = x.$$

So Bob chooses the decryption function $d_r(y) = y \oplus r$.

One-time pad (cont'd)

How should Alice and Bob choose r for this scheme to be secure?

They should pick r at random, flipping a coin for each bit, so that the resulting string is equally likely to be any element of $\{0,1\}^n$.

This will ensure that if Eve intercepts the encoded message $y = e_r(x)$, she gets no information about x: all r's are equally possible, thus all possibilities for x are equally likely!

The downside of one-time pad

The downside of the one-time pad is that it has to be discarded after use, hence the name.

A second message encoded with the same pad would not be secure, because if Eve knew $x \oplus r$ and $z \oplus r$ for two messages x and z, then she could take the exclusive-or to get $x \oplus z$, which might be important information:

1. it reveals whether the two messages begin or end the same;
2. if one message contains a long sequence of zeros (as could easily be the case if the message is an image), then the corresponding part of the other message will be exposed.

Therefore the random string that Alice and Bob share has to be the combined length of all the messages they will need to exchange.

Random strings are costly!

The Rivest-Shamir-Adelman (RSA) cryptosystem

An example of **public-key cryptography:**

- Anybody can send a message to anybody else using publicly available information, rather like addresses or phone numbers.
- Each person has a public key known to the whole world and a secret key known only to him- or herself.
- When Alice wants to send message x to Bob, she encodes it using his *public key*.
- Bob decrypts it using his *secret key*, to retrieve x.
- Eve is welcome to see as many encrypted messages for Bob as she likes, but she will not be able to decode them, *under certain simple assumptions*.

Property

Pick any two primes p and q and let $N = pq$. For any e relatively prime to $(p-1)(q-1)$:

1. The mapping $x \mapsto x^e \mod N$ is a bijection on $\{0, 1, \ldots, N-1\}$.
2. The inverse mapping is easily realized: let d be the inverse of e modulo $(p-1)(q-1)$. Then for all $x \in \{0, 1, \ldots, N-1\}$,

$$(x^e)^d \equiv x \pmod N.$$

- The mapping $x \mapsto x^e \mod N$ is a reasonable way to encode messages x; no information is lost. So, if Bob publishes (N, e) as his **public key**, everyone else can use it to send him encrypted messages.
- Bob should retain the value d as his **secret key**, with which he can decode all messages that come to him by simply raising them to the dth power modulo N.

Proof of the property

If the mapping $x \mapsto x^e \mod N$ is invertible, it must be a bijection; hence statement 2 implies statement 1.

To prove statement 2, we start by observing that e is invertible modulo $(p-1)(q-1)$ because it is relatively prime to this number. It remains to show that

$$(x^e)^d \equiv x \mod N.$$

Since $ed \equiv 1 \mod (p-1)(q-1)$, we can write

$$ed = 1 + k(p-1)(q-1)$$

for some k. Then

$$(x^e)^d - x = x^{ed} - x = x^{1+k(p-1)(q-1)} - x.$$

$x^{1+k(p-1)(q-1)} - x$ is divisible by p (since $x^{p-1} \equiv 1 \pmod p$) and likewise by q. Since p and q are primes, this expression must be divisible by $N = pq$. $\square$

RSA protocol

Bob chooses his public and secret keys:

1. He starts by picking two large (n-bit) random primes p and q.
2. His public key is (N, e) where $N = pq$ and e is a $2n$-bit number relatively prime to $(p-1)(q-1)$. A common choice is $e = 3$ because it permits fast encoding.
3. His secret key is d, the inverse of e modulo $(p-1)(q-1)$, computed using the extended Euclid algorithm.

Alice wishes to send message x to Bob:

1. She looks up his public key (N, e) and sends him $y = (x^e \mod N)$, computed using an efficient modular exponentiation algorithm.
2. He decodes the message by computing $y^d \mod N$.

Security assumption for RSA

The security of RSA hinges upon a simple assumption:

Given N, e, and $y = x^e \mod N$, it is computationally intractable to determine x.

How might Eve try to guess x? She could experiment with all possible values of x, each time checking whether $x^e \equiv y \mod N$, but this would take exponential time.

Or she could try to factor N to retrieve p and q, and then figure out d by inverting e modulo $(p-1)(q-1)$, but we believe *factoring to be hard*.

Universal Hashing

Motivation

We will give a short "nickname" to each of the 2^{32} possible IP addresses.

You can think of this short name as just a number between 1 and 250 (we will later adjust this range very slightly).

Thus many IP addresses will inevitably have the same nickname; however, we hope that most of the 250 IP addresses of our particular customers are assigned distinct names, and we will store their records in an array of size 250 indexed by these names.

What if there is more than one record associated with the same name?

Easy: each entry of the array points to a linked list containing all records with that name. So the total amount of storage is proportional to 250, the number of customers, and is independent of the total number of possible IP addresses.

Moreover, if not too many customer IP addresses are assigned the same name, lookup is fast, because the average size of the linked list we have to scan through is small.

Hash tables

How do we assign a short name to each IP address?

This is the role of a hash function: A function h that maps IP addresses to positions in a table of length about 250 (the expected number of data items).

The name assigned to an IP address x is thus $h(x)$, and the record for x is stored in position $h(x)$ of the table.

Each position of the table is in fact a *bucket*, a linked list that contains all current IP addresses that map to it.
Hopefully, there will be very few buckets that contain more than a handful of IP addresses.

How to choose a hash function?

In our example, one possible hash function would map an IP address to the 8-bit number that is its last segment:

$$h(128.32.168.80) = 80.$$

A table of $n = 256$ buckets would then be required.

But is this a good hash function?

Not if, for example, the last segment of an IP address tends to be a small (single- or double-digit) number; then low-numbered buckets would be crowded.

Taking the first segment of the IP address also invites disaster, for example, if *most of our customers come from Asia*.

How to choose a hash function? (cont'd)

- There is nothing *inherently wrong* with these two functions. If our 250 IP addresses were uniformly drawn from among all $N = 2^{32}$ possibilities, then these functions would behave well.
  The problem is we have no guarantee that the distribution of IP addresses is *uniform*.
- Conversely, there is no *single hash function*, no matter how sophisticated, that behaves well on all sets of data.
  Since a hash function maps 2^{32} IP addresses to just 250 names, there must be a collection of at least

  $$2^{32}/250 \approx 2^{24} \approx 16,000,000$$

  IP addresses that are assigned the same name (or, in hashing terminology, **collide**).

Solution: *let us pick a hash function at random from some class of functions.*

Families of hash functions

Let us take the number of buckets to be not 250 but $n = 257$. a prime number!
We consider every IP address x as a quadruple $x =$

$$(x_1, x_2, x_3, x_4)$$

of integers modulo n.

We can define a function h from IP addresses to a number $\mod n$ as follows:

Fix any four numbers mod $n = 257$, say 87, 23, 125, and 4. Now map the IP address $(x_1, \ldots, x_4)$ to $h(x_1, \ldots, x_4) = (87x_1 + 23x_2 + 125x_3 + 4x_4) \mod 257$.

In general for any four coefficients $a_1, \ldots, a_4 \in \{0, 1, \ldots, n-1\}$ write $a = (a_1, a_2, a_3, a_4)$ and define h_a to be the following hash function:

$$h_a(x_1, \ldots, x_4) = (a_1 \cdot x_1 + a_2 \cdot x_2 + a_3 \cdot x_3 + a_4 \cdot x_4) \mod n.$$

Property

Consider any pair of distinct IP addresses $x = (x_1, \ldots, x_4)$ and $y = (y_1, \ldots, y_4)$. If the coefficients $a = (a_1, \ldots, a_4)$ are chosen uniformly at random from $\{0, 1, \ldots, n-1\}$, then

$$\Pr\left[h_a(x_1, \ldots, x_4) = h_a(y_1, \ldots, y_4)\right] = \frac{1}{n}.$$

Universal families of hash functions

Let

$$\mathcal{H} = \left\{h_a \mid a \in \{0, 1, \ldots, n-1\}^4\right\}.$$

It is **universal**:

> *For any two distinct data items x and y, exactly $|\mathcal{H}|/n$ of all the hash functions in $\mathcal{H}$ map x and y to the same bucket, where n is the number of buckets.*