Chapter 1

Numbers and Arithmetic

Processing, display, and communication of digital information, that is, information encoded as numbers, is accomplished by various kinds of arithmetic with various kinds of numbers. *Algorithms* to do this are sequences of operations such as reading and writing digits, addition and multiplication, comparisons and logical decisions. Only finite algorithms can be used: these are procedures in which

- Every operation can be performed in a finite time;
- The algorithm is guaranteed to stop, with an answer, after a finite number of operations.

For an algorithm to be finite, its arithmetic operations can only be carried out to *finite precision*, namely on finitely many digits. In reality, a computer can keep only a small number of digits for each number because memory, processing and data communication are costly resources. This usually poses no problems since the digital information of multimedia signals is itself of low precision. For example, a "CD-quality" digital sound recording consists of a sequence of numbers measuring the electrical output of a microphone at sequential times, with a precision of five decimal digits or less per measurement. Images from typical scanners are even less precise, consisting of arrays of numbers measuring light intensity to three decimal digits. Physical measurements always have some imprecision or measurement error which is costly to reduce, and human senses cannot make use of much precision anyway. The result is that computation for multimedia signal processing can be done with low precision arithmetic.

Most computers distinguish between *integers* and floating-point numbers or *floats*, which are approximations to real numbers. Either class is suitable for representing finite-precision information. Floats have the advantage of using all the precision available with given memory space, independent of their magnitude. Integers are superior for certain exact calculations such as counting.

All computers have a fixed range of *representable values* for both integers and floats, and have efficient circuitry for arithmetic with numbers in those ranges. For

example, most computers can perform floating-point arithmetic very efficiently at some built-in fixed precision such as seven or 14 decimal digits. More precision is obtained when needed using software algorithms that are consequently less efficient.

This chapter will examine some of the mathematical properties of integer and floating-point arithmetic, develop a few facts about integer arithmetic that are useful for secret codes, define precision and analyze the propagation of error caused by low-precision floating-point computation, and describe a standard computer representation of floats with an explanation of how it aids efficient computation with good control of error propagation.

1.1 Integers

The set of *integers* is denoted by $\mathbf{Z} \stackrel{\text{def}}{=} \{\dots,-2,-1,0,1,2,3,\dots\}$. The use of the letter 'Z' derives from the German noun "Zahl," or "number." $\mathbf{Z}$ is a *disjoint union* of three subsets: the *positive integers* $\mathbf{Z}^+ \stackrel{\text{def}}{=} \{x \in \mathbf{Z} : x > 0\} = \{1,2,3,\dots\}$, the *negative integers* $\mathbf{Z}^- \stackrel{\text{def}}{=} -\mathbf{Z}^+ = \{x \in \mathbf{Z} : x < 0\} = \{-1,-2,-3,\dots\}$, and zero: $\mathbf{Z} = \mathbf{Z}^- \cup \{0\} \cup \mathbf{Z}^+$. The *natural numbers* $\mathbf{N} \stackrel{\text{def}}{=} \mathbf{Z}^+ \cup \{0\} = \{x \in \mathbf{Z} : x \geq 0\} = \{0,1,2,\dots\}$ are another disjoint union from these pieces.

Although $\mathbf{Z}$, $\mathbf{Z}^+$, $\mathbf{Z}^-$ and $\mathbf{N}$ are all infinite sets, any individual integer is determined by a finite list of symbols. The choice of symbols is largely a matter of convenience. Humans use arabic numerals that can be manipulated easily on paper and by calculator. Computers use sequences of low and high voltages, nominally 0s and 1s, that can be processed electronically at nearly the speed of light.

Division and greatest common divisors

For any integers a, b with $b \neq 0$, there are unique integers q and r, called the *quotient* and *remainder*, respectively, satisfying the *division properties*:

1. $a = qb + r$;
2. $0 \leq r < |b|$.

If $a, b \in \mathbf{N}$, then $q, r \in \mathbf{N}$ as well. Quotient q is given by *integer division* $q = \lfloor \frac{a}{b} \rfloor$, where the *floor* function $\lfloor x \rfloor$ computes "the greatest integer less than or equal to x." Remainder r is the leftover: $r = a - qb$.

The "long division" algorithm can be used to determine q and r from a and b. Integer division takes finitely many steps: if a and b have at most N digits, then computing q and r requires $O(N^2)$ one-digit operations[1] such as trial multiplications or subtractions.

The Standard C programming language has special integer quotient and remainder operators / and % for computing q and r, as seen in this fragment of a computer program:

[1] See Appendix B.3, page 289, for an explanation of this "big-Oh" notation.

```
int a, b, q, r;
 ...
q = a/b;
r = a%b;
 ...
```

Standard C guarantees that conditions 1 and 2 hold for $a \geq 0$ and $b > 0$, but, unfortunately, condition 2 is not guaranteed if a or b is negative. For example, one typical machine computes as follows:

a	b	q=a/b	r=a%b
17	5	3	2
17	-5	-3	2
-17	5	-3	-2
-17	-5	3	-2

If $b \neq 0$ and the remainder upon dividing a by b is zero, then we say[2] "*b divides a*" and write $b|a$. Some facts written with this notation are:

- $b|0$: any b divides $a = 0$ (take $q = 0$ and $r = 0$ in $a = qb + r$);
- $1|a$: $b = 1$ divides any a (take $q = b$ and $r = 0$ in $a = qb + r$);
- $b|a \Rightarrow \pm b|\pm a$: if b divides a then $\pm b$ divides $\pm a$ (if $a = qb$, then $-a = (-q)b$, $-a = q(-b)$, and $a = (-q)(-b)$).

Lemma 1.1 *If $a > 0$ and b divides a, then $0 < |b| \leq a$.*

Proof: From $b|a$ we have $a = qb$, so $0 < a = |a| = |qb| = |q||b|$, so $|q| \neq 0$, so $|q| \geq 1$. Thus $a - |b| = (|q| - 1)|b| \geq 0$, so $a \geq |b|$. Finally, $b \neq 0$ since $qb = a \neq 0$, so $|b| > 0$. □

A positive integer c is said to be the *greatest common divisor* of two integers a and b if

gcd-1: c divides both a and b: $c|a$ and $c|b$;

gcd-2: Any integer that divides both a and b also divides c: $(d|a$ and $d|b) \Rightarrow d|c$.

If it exists, it must be unique by property gcd-2 and Lemma 1.1. The proof is that if c_1 and c_2 are both greatest common divisors of a and b, then $c_1|c_2$ so $c_1 \leq c_2$, and also $c_2|c_1$ so $c_2 \leq c_1$. Thus $c_2 = c_1$. But existence is guaranteed, too:

Theorem 1.2 *Every pair of integers a, b, not both zero, has a greatest common divisor, which can be written as $xa + yb$ for some integers x, y.*

[2] Or else we say "b is a *divisor* of a," or "a is *divisible* by b."

Proof: Let $D = \{xa + yb : x, y \in \mathbf{Z}\}$. D contains some nonzero integers since not both a and b are zero, so D must contain some positive integers since $d \in D \Rightarrow -d \in D$. Let $c = x_0a + y_0b$ be the smallest positive integer[3] in D. Then any integer z that divides both a and b will divide c, since $a = nz$ and $b = mz$ implies $c = (x_0n + y_0m)z$. Hence c satisfies property gcd-2.

To show that c divides a, write $a = qc + r$ with $0 \leq r < c$. Then $a = q(x_0a + y_0b) + r$, so $r = (1 - qx_0)a + (-qy_0)b \in D$. This r must be zero since otherwise it would be a smaller positive element of D than c. Hence $c|a$. The same argument shows that c divides b, so c satisfies property gcd-1. □

We may use the functional notation $c = \gcd(a, b)$ for this unique greatest common divisor. For example, $\gcd(-12, 16) = 4$ and $\gcd(256\,964\,964, 6\,447\,287) = 73$. Note that $\gcd(0, 0)$ is undefined since every integer, no matter how large, divides both zeroes. Hence, the "not both zero" assumption is necessary.

By convention, $\gcd(a, 0) = \gcd(0, a) = |a|$ for any $a \neq 0$. Other useful facts are:

- $\gcd(a, b) = \gcd(b, a) = \gcd(|a|, |b|)$.
- If $a' = \max(|a|, |b|)$ and $b' = \min(|a|, |b|)$, then $\gcd(a, b) = \gcd(a', b')$.
- If $a \neq 0$ and $b \neq 0$, then $\gcd(a, b) \leq \min(|a|, |b|)$.
- If a divides b, then $\gcd(a, b) = |a|$.

An efficient algorithm for computing greatest common divisors has been known at least since Euclid wrote it down thousands of years ago, having first observed:

Corollary 1.3 *If a and b are integers with $a \neq 0$, then* $\gcd(a, b) = \gcd(b, a\%b)$.

Proof: Any common divisor c of a and b is also a common divisor of b and $r = a\%b$, since $a = nc$ and $b = mc$ with $a = qb + r$ implies $r = a - qb = (n - qm)c$. Taking the largest c shows that $\gcd(a, b) \leq \gcd(b, a\%b)$.

Conversely, by Theorem 1.2 there are integers x, y such that $\gcd(b, a\%b) = xb + y(a\%b) = xb + y(a - qb) = ya + (x - qy)b$. Thus $\gcd(b, a\%b)$ is a positive element of the set D defined in the proof of that theorem. It is no smaller than the least positive element of D, which is $\gcd(a, b)$, so $\gcd(a, b) \leq \gcd(b, a\%b)$.

Combining the two inequalities gives $\gcd(a, b) = \gcd(b, a\%b)$. □

Applying this corollary repeatedly gives $\gcd(a, b) = \cdots = \gcd(c, 0) = |c|$, which may be implemented by a *recursive function*, namely one that calls itself. To start it off, first prepare the inputs by replacing $a \leftarrow \max(|a|, |b|)$ and $b \leftarrow \min(|a|, |b|)$, so as to guarantee that $a > 0$, $b \geq 0$, and $a \geq b$:

Euclid's Algorithm

```
gcd( a, b ):
[0] If b==0 then return a
[1] Else return gcd( b, a%b )
```

[3]We take for granted the *Least Element Principle*: any subset of $\mathbf{Z}^+$ has a smallest element.

Notice that `gcd()` first tests a *termination condition*, which in this case is a comparison of b with zero. If that condition is satisfied then the algorithm ends with an answer. Otherwise, it calls itself[4] with new arguments that are closer to satisfying the termination condition.

To analyze this algorithm, let a_n, b_n be the respective values of a, b at the nth call to `gcd()`. Then $a_1 = a$, $b_1 = b$, $a_{n+1} = b_n$ and $b_{n+1} = a_n\%b_n$ for $n = 1, 2, \ldots$ The remainder property $b > a\%b$ insures that $b = b_1 > b_2 > b_3 > \cdots \geq 0$, and since each b_n is an integer, the recursion must terminate after at most b calls. But each function call requires copying the digits of numbers no bigger than a, the division algorithm, and reading the digits of a number to see if they are all 0. Hence, each step takes finitely many calculations, so the algorithm is finite.

Suppose $k \geq 1$ is the least index for which $b_k = 0$, the termination condition. Then the returned value a_k is the greatest common divisor of a_k and $b_k = 0$. This return value passes up through all the recursively-called `gcd()` functions including the first. By Corollary 1.3, it is the greatest common divisor of the original pair a, b, so Euclid's algorithm returns the correct answer.

The preceding finiteness analysis shows that `gcd(a,b)` terminates after $O(b)$ recursive function calls, but in fact there is a far better efficiency estimate for large b. Since $b_{n+1} < b_n$ for all $n = 1, 2, \ldots$, we may write $b_{n+1} = b_n - d_n$ for some $0 < d_n \leq b_n$. But also, $a_{n+1} = b_n$, so for any $1 \leq n \leq k-2$, $b_{n+2} = a_{n+1}\%b_{n+1} = b_n\%(b_n - d_n)$, which implies two things: $b_{n+2} < b_{n+1} = b_n - d_n$, and also $b_{n+2} = b_n\%(b_n - d_n) \leq d_n$. Thus $b_{n+2} \leq \min\{b_n - d_n, d_n\} \leq \frac{1}{2}b_n$, so b_n is halved every two function calls. Thus the number of recursions required by Euclid's algorithm is at most $2\log_2 b = \log_{\sqrt{2}} b$, which is proportional to the number of digits in b rather than the much larger value of b itself.

*A better complexity estimate for gcd

A somewhat better speed estimate follows from a worst-case analysis done almost two centuries ago using the *Fibonacci numbers*, a sequence of integers defined recursively by

$$F_0 = 0; \qquad F_1 = 1; \qquad F_{n+2} = F_{n+1} + F_n, \quad n = 0, 1, 2, \ldots \tag{1.1}$$

For example $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, and $F_9 = 34$.

Suppose $a \geq b \geq 0$ with not both $a = 0$ and $b = 0$. Before bringing in the Fibonacci numbers we may dispose of two trivial cases:

- If $a > b = 0$, then $\gcd(a, 0) = a$ and Euclid's algorithm terminates after one function call.
- If $a = b > 0$, then $a\%b = 0$ so $\gcd(a, b) = \gcd(b, 0) = b$ and Euclid's algorithm terminates after two function calls.

This leaves the main case $a > b > 0$:

[4] Recursively!

Theorem 1.4 (Gabriel Lamé, 1844) *Suppose $a > b > 0$ and Euclid's algorithm terminates after k function calls. Then $a \geq F_{k+1}$ and $b \geq F_k$, where F_k is the kth Fibonacci number.*

Proof: Use induction on k. Case $k = 1$ requires $b = 0$ and is inapplicable; it is the first trivial case handled previously. Case $k = 2$ must have $a\%b = 0$ so that the first recursive call to `gcd()` ends immediately with the termination condition. The smallest pair $a > b > 0$ for which this happens is $a = 2 = F_3 = F_{k+1}$ and $b = 1 = F_2 = F_k$.

Now suppose that the result is true for k and let $a > b > 0$ be a pair that requires exactly $k + 1$ function calls. Then the first recursive function call gets the arguments $b, a\%b$, which will require exactly k more recursive calls, so $b \geq F_{k+1}$ and $a\%b \geq F_k$. But then the integer division properties imply

$$a = (a/b)b + (a\%b) \;\geq\; (a/b)F_{k+1} + F_k \;\geq\; F_{k+1} + F_k = F_{k+2},$$

since $a/b \geq 1$. Hence the inductive hypothesis holds for $k + 1$. □

To complete the estimate we use the formula for the nth Fibonacci number:

$$F_n = \frac{\varphi_+^n - \varphi_-^n}{\sqrt{5}}, \qquad \varphi_+ \stackrel{\text{def}}{=} \frac{1+\sqrt{5}}{2} \approx 1.618, \quad \varphi_- \stackrel{\text{def}}{=} \frac{1-\sqrt{5}}{2} \approx -0.618,$$

The two numbers $\varphi_\pm$ are the distinct roots of the quadratic *characteristic equation* $r^2 = r + 1$ associated to the recursion in Equation 1.1. The power φ_+^n dominates the magnitude of F_n since $|\varphi_-^n| \leq 1$ for all $n \in \mathbf{N}$, so

$$F_n \geq \frac{\varphi_+^n - 1}{\sqrt{5}},$$

which means that $\gcd(a, b)$ with $a > b > 0$ will require k function calls only if $b \geq (\varphi_+^k - 1)/\sqrt{5}$. Taking logarithms and ignoring constant terms shows that computing $\gcd(a, b)$ will take about $\log_{\varphi_+} b \approx \log_{1.618} b$ function calls. This is a somewhat better estimate than $\log_{\sqrt{2}} b \approx \log_{1.414} b$. However, both estimates are $O(\log b)$ and differ only slightly in the constants of proportionality.

Primes and unique factorization

Integers a and b are called *relatively prime* if $\gcd(a, b) = 1$. Any integer a is relatively prime to $b = 1$.

Lemma 1.5 *If c divides ab and $\gcd(a, c) = 1$, then c divides b.*

Proof: Write $1 = \gcd(a, c) = m_0 a + n_0 c$ as in the proof of Theorem 1.2. Then $b = m_0 ab + n_0 cb$. Since c evidently divides $n_0 cb$, and c divides $m_0 ab$ by assumption, it follows that c divides $m_0 ab + n_0 cb = b$. □

An integer $p > 1$ is called *prime* if its only divisors are ± 1 and $\pm p$. Thus, if a is any integer and p is prime, either p divides a or else a and p are relatively prime.

It follows from Lemma 1.5 that if p is prime and p divides ab, then either p divides a or p divides b.

It is an easy exercise to prove from these definitions that distinct primes are relatively prime.

Suppose that $N > 1$ is a fixed integer. Then either N is prime or there is some $1 < a < N$ that divides N. The same argument may be repeated for $N_1 = a$ and $N_2 = N/a$, both of which are positive integers strictly less than $N = N_1 N_2$. This recursive decomposition terminates after finitely many steps at the *prime divisors* $p_1, p_2, \ldots, p_k$ of N.

The smallest primes are 2, 3, 5, 7, 11, 13, and 17, but the list never ends:

Theorem 1.6 *There are infinitely many primes.*

Proof: Let $p_1, p_2, \ldots, p_n$ be any finite set of primes. Then $N \stackrel{\text{def}}{=} 1 + p_1 p_2 \cdots p_n$ is relatively prime to all of them. But either N is prime and not in the set, or else N contains a prime divisor not in the set. Hence no finite set of primes can contain all primes. Conclude that the set of primes is infinite. □

To determine whether an integer N is prime may be done by trial division with all integers less than or equal to $\sqrt{N}$. This is very slow for large N and there are more sophisticated *primality tests* which are faster.

The prime divisors of an integer $N > 1$ yield its *prime factorization*[5] $N = p_1 p_2 \cdots p_k$.

Theorem 1.7 *Prime factorization is unique: If $p_1 \cdots p_n = q_1 \cdots q_m$ and the p's and q's are primes, then $n = m$ and, possibly after re-indexing, the p's are the same as the q's.*

Proof: Let r be one of the primes $\{q_1, q_2, \ldots, q_m\}$. Since r divides $p_1 \cdots p_n$, it must divide one of the p's. But then r must equal one of the p's since two primes are either equal or relatively prime. Thus the set of p's includes all the q's. Similarly, the set of q's includes all the p's. If a prime r appears i times in one factorization and $j > i$ times in the other, then dividing both by r^i leaves equal factorizations with $j - i > 0$ factors r in one but no factor r in the other, which is not possible. Thus the count of each prime must be the same in both factorizations. □

Unique factorization requires that 1 not be considered prime. Computing the prime factorization of a large integer cannot be done fast by any known method, and this tough problem can be used for *cryptography.*

1.1.1 Modular arithmetic

Fix an integer $M > 1$ and say that two integers a and b are *congruent modulo M* if M divides $b - a$, that is, if a and b differ by a multiple of M. Such a condition is written $a \equiv b \pmod{M}$.

[5] Some prime divisors may appear more than once.

Congruent numbers must leave the same remainders $a\%M$ and $b\%M$, so the finite set $\{n : 0 \leq n < M\} = \{0, 1, \ldots, M-1\}$, which may also be called M, contains one representative from each of the *congruence classes* modulo M. Namely, every integer is congruent to one of the numbers $0, 1, \ldots$, or $M-1$, modulo M.

Modular addition, subtraction, and multiplication is similar to ordinary arithmetic except that equality is replaced by congruence. Thus the answer need only be determined within an integer multiple of the modulus M, and the operands can be replaced by congruent representatives from the set M:

Lemma 1.8 *If $a, b, c \in \mathbf{Z}$ are respectively congruent to $\alpha, \beta, \gamma \in M$ modulo M, then*

$$ab + c \equiv \alpha\beta + \gamma \pmod{M}.$$

Proof: Write $a = \alpha + xM$, $b = \beta + yM$, and $c = \gamma + zM$, where x, y, and z are integers. Then $ab + c = \alpha\beta + \gamma + (\alpha y + \beta x + z)M$, so $ab + c - (\alpha\beta + \gamma)$ is an integer multiple of the modulus M. □

The modular additive inverse of x is any number y such that $x + y \equiv 0 \pmod{M}$. For $x \in M$, a natural candidate is $y = M - x$, which also belongs[6] to the set M. Thus we can mimic ordinary signed integers in modular arithmetic by considering numbers between 0 and $M/2$ to be positive, and those between $M/2$ and M to be negative. Modular addition and subtraction will then agree with ordinary addition and subtraction for all operations with integers of sufficiently small magnitude. If $|x| < M/4$ and $|y| < M/4$, then $x + y$ will have the same representative in $\mathbf{Z}$ as in the signed interpretation of the set M.

Modular multiplication likewise agrees with ordinary integer multiplication of small enough integers. Where $M/4$ was a magnitude limit for addition, it is $\sqrt{M/2}$ for multiplication.

Modular division b/a can sometimes be done even if a does not divide b. For example, $5 \cdot 2 \equiv 1 \pmod{9}$, so we can write $1/2 \equiv 5 \pmod{9}$ or $1/5 \equiv 2 \pmod{9}$. Define a *quasi-inverse* of a modulo M to be any integer a' satisfying

$$aa' \equiv 1 \pmod{M}. \tag{1.2}$$

This is a multiplicative inverse in modular arithmetic, but it has no analog in ordinary integer arithmetic.

Lemma 1.9 *Let $M > 1$ and a be integers. Then a has quasi-inverse a' modulo M if and only if $\gcd(a, M) = 1$, and in that case a' is unique in the set $\{1, \ldots, M-1\}$.*

Proof: If $\gcd(a, M) = 1$, then write $1 = \gcd(a, M) = m_0 a + n_0 M$ as in the proof of Theorem 1.2. Evidently m_0 is a quasi-inverse of a, and if $m_0 \geq M$ or $m_0 < 0$, an appropriate multiple of M can be added to get a quasi-inverse $a' = m_0 + kM \in \{0, 1, 2, \ldots, M-1\}$. But this a' cannot be zero since $a \cdot 0 \equiv 0 \not\equiv 1 \pmod{M}$. For uniqueness, note that if both a' and a'' satisfy $aa' \equiv aa'' \equiv 1 \pmod{M}$, then M

[6] What about $x = 0$?

divides $a(a' - a'')$. By Lemma 1.5, M must divide $a' - a''$, so if both a' and a'' lie in the set $\{1, \ldots, M-1\}$ they must be equal.

On the other hand, if $\gcd(a, M) = m_0 a + n_0 M = d > 1$, then there are no integers x, y such that $0 < ax + My < d$. In particular, there are none that give $ax = 1 + yM$, so there is no integer x solving $ax \equiv 1 \pmod{M}$. □

Corollary 1.10 *If M is prime, then every integer a in the set $\{1, \ldots, M-1\}$ has a quasi-inverse modulo M.* □

The following extension of Euclid's algorithm, from Knuth's *Fundamental Algorithms*, page 14, finds quasi-inverses. Given two positive integers a and b, it computes $d = \gcd(a, b)$ and two integers x, y satisfying $ax + by = d$:

Extended Euclid's Algorithm

```
gcdx( a, b ):
[0] Initialize x=0, y=1, xo=1, yo=0, c=a, and d=b
[1] Let q = c/d and r = c%d
[2] If r==0, then go to [5]
[3] Let c = d, d = r, t = xo, xo = x, x = t-q*x,
          t = yo, yo = y, and y = t-q*y
[4] Go to [1]
[5] Print x, y, and d
```

Starting with relatively prime a and $b = M$, the output will be a quasi-inverse x of a, some integer y, and the known result $d = \gcd(a, M) = 1$. After storing $k = \lfloor x/M \rfloor$, we may adjust $x \leftarrow x - kM$ to get a quasi-inverse in $\{1, \ldots, M-1\}$. Note that this requires adjusting $y \leftarrow y + ka$ to preserve the equality $ax + by = d$.

Modular exponentiation and Euler's totient

For each integer $M > 0$, the set $\{k \in \mathbf{N} : \gcd(k, M) = 1\}$ is preserved under multiplication and quasi-inversion modulo M: $\gcd(k_1, M) = 1$ and $\gcd(k_2, M) = 1$ implies $\gcd(k_1 k_2, M) = 1$ by Lemma 1.5 applied to any common prime divisor p of M and $k_1 k_2$, and $\gcd(k, M) = 1$ implies k has a quasi-inverse k' which implies k' has a quasi-inverse k which implies $\gcd(k', M) = 1$ by Lemma 1.9. Evidently $\gcd(1, M) = 1$, and thus the finite set of remainders

$$G_M \stackrel{\text{def}}{=} \{k\%M : k \in \mathbf{N}, \gcd(k, M) = 1\} \subset \{1, \ldots, M-1\}$$

contains 1 and is preserved under multiplication and quasi-inversion modulo M. Such a set is called a *group*. The number of elements in a group is called its *order*.

Definition 1 Euler's totient function $\phi(M)$ *is the order of G_M, namely the number of integers in $\{1, \ldots, M-1\}$ which are relatively prime to M and are therefore quasi-invertible modulo M.*

Subsets of G_M that contain 1 and are also preserved under multiplication and quasi-inversion modulo M are called *subgroups*. For example, the *cyclic subgroup* generated by $a \in G_M$ consists of the remainders $\{a^k\%M : k \in \mathbf{N}\}$. This subset evidently contains $1 = a^0$ and is closed under multiplication: $a^p a^q = a^{p+q}$. The order of the cyclic subgroup of G_M generated by a is also called the *order of* a and is denoted $\omega(a)$.

Lemma 1.11 *The order of a in G_M is the smallest positive integer k such that $a^k \equiv 1 \pmod{M}$.*

Proof: Let k be the smallest positive integer such that $a^k \equiv 1 \pmod{M}$. Then for any $0 < p < q \le k$ we must have $a^p \not\equiv a^q \pmod{M}$ for otherwise $a^{q-p} \equiv 1 \pmod{M}$ with $0 < q - p < k$. Hence the k remainders $a\%M, a^2\%M, \ldots, a^k\%M$ are distinct, so $k \le \omega(a)$.

For the reverse inequality, note that $p = q + nk$ implies

$$a^p \equiv a^{q+nk} \equiv a^q a^{nk} \equiv a^q (a^k)^n \equiv a^q (1)^n \equiv a^q \pmod{M},$$

so that $a^p \equiv a^q \pmod{M}$ if $p \equiv q \pmod{k}$. Hence there are at most k distinct values in $\{a^p\%M : p \in \mathbf{N}\}$, so $k \ge \omega(a)$. Combining inequalities shows that $k = \omega(a)$. □

Lemma 1.11 shows that the cyclic subgroup generated by a is closed under quasi-inversion: for any $p \in \mathbf{N}$ there is some $n \in \mathbf{N}$ such that $nk - p \in \mathbf{N}$. But then a^p has quasi-inverse a^{nk-p} since

$$a^p\, a^{nk-p} \equiv a^{nk} \equiv (a^k)^n \equiv (1)^n \equiv 1 \pmod{M}.$$

Theorem 1.12 (Lagrange) [7] *The order of a in G_M divides $\phi(M)$.*

Proof: Denote by H the cyclic subgroup of G_M generated by a, and define the *coset*

$$gH = \{gh\%M : h \in H\}$$

determined by each $g \in G_M$. Note first that for elements $g_1, g_2 \in G_M$, either $g_1 H = g_2 H$ or $g_1 H \cap g_2 H = \emptyset$. The first occurs if $g_2' g_1 \in H$, for then $g_1 x \in g_1 H$ is the same as $g_2 y \in g_2 H$ with $y \stackrel{\text{def}}{=} g_2' g_1 x \in H$, using the fact that H is closed under multiplication. The second occurs if $g_2' g_1 \notin H$ because if $g_1 H$ and $g_2 H$ shared an element $z \equiv g_1 x \equiv g_2 y \pmod{M}$, some $x, y \in H$, it would mean

$$g_1 x \equiv g_2 y \pmod{M} \Rightarrow g_2' g_1 \equiv y x' \pmod{M} \Rightarrow g_2' g_1 \in H,$$

since H is closed under quasi-inversion. Hence no shared element can exist.

Next, note that each element $g \in G_M$ is in some coset, in fact in gH, simply because $1 \in H$. Thus $G_M = \bigcup_{g \in G_M} gH$, and this union is disjoint.

[7] Lagrange actually proved that the number of elements in a subgroup divides the number of elements in the group, but only this special case is needed here.

Finally, note that every coset has the same number of elements $\omega(a)$ as H, since the mapping $H \to gH$ given by $x \mapsto gx$ is a one-to-one correspondence with inverse $y \mapsto g'y$. Combining these observations shows that a whole number multiple of $\omega(a)$ gives the order $\phi(M)$ of G_M. □

Euler's totient satisfies a modular exponentiation identity:

Theorem 1.13 (Euler) *If M and a are any integers with* $\gcd(a, M) = 1$, *then* $a^{\phi(M)} \equiv 1 \pmod{M}$.

Proof: By Lemma 1.11 and Theorem 1.12, the smallest $k > 0$ such that $a^k \equiv 1 \pmod{M}$ is $\omega(a)$ and divides $\phi(M)$, the number of elements in G_M. We may thus write $\phi(M) = nk$ for some $n \in \mathbf{N}$. But then

$$a^{\phi(M)} \equiv a^{nk} \equiv (a^k)^n \equiv (1)^n \equiv 1 \pmod{M},$$

proving the result. □

The special case of prime M, for which $\phi(M) = M - 1$, gives:

Corollary 1.14 (Fermat) [8] *If M is prime and $0 < a < M$, then $a^{M-1} \equiv 1 \pmod{M}$.* □

RSA encryption and decryption

Now suppose that e, d are quasi-inverses modulo $\phi(M)$, so $ed = 1 + n\phi(M)$ for some $n \in \mathbf{N}$. Then $a \in G_M$ can be recovered from $a^e \pmod{M}$ as follows:

$$(a^e)^d \equiv a^{ed} \equiv a^{1+n\phi(M)} \equiv (a)(a^{\phi(M)})^n \equiv a \pmod{M}$$

This idea underpins the *Rivest-Shamir-Adleman (RSA)* encryption and decription algorithms. Given public information M and e it is easy to encrypt a message represented by the *cleartext* number a as the *cyphertext* number $c = a^e \% M$. The recipient can recover $a = c^d \% M$ from the cyphertext using secret information d and public information M. Both modular exponentiations can be performed efficiently using the `modular_power()` function defined further on.

An eavesdropper wanting to recover cleartext a from intercepted cyphertext c must know $\phi(M)$ in order to compute the decryption exponent d, but this is equivalent to factoring M into primes:

Theorem 1.15 *Given the prime factorization $M = p_1^{m_1} \cdots p_n^{m_n}$, where $\{p_k : k = 1, \ldots, n\}$ are distinct primes and $\{m_k : k = 1, \ldots, n\}$ are positive integers, we have*

$$\phi(M) \stackrel{\text{def}}{=} \#\{k \in M : \gcd(k, M) = 1\} = \prod_{i=1}^{n} \left(p_i^{m_i} - p_i^{m_i - 1}\right),$$

where the symbol $\prod$ denotes the product of the terms that follow.

---

[8] This result is often called Fermat's Little Theorem in contrast to his Great or Last Theorem, proved only after a 350 year effort, that no positive integers x, y, z can solve $x^n + y^n = z^n$ if $n > 2$.

Proof: First note that $\phi(p) = p - 1$ for any prime number p. Similarly, compute $\phi(p^m) = p^m - p^{m-1}$ for any individual prime p and any positive integer power m since the only numbers in $\{0, 1, 2, \ldots, p^m - 1\}$ which are not relatively prime to p^m are the multiples of p: $0p, 1p, \ldots, (p^{m-1} - 1)p = p^m - p$, of which there are evidently p^{m-1}.

Next, note that if $\gcd(M, N) = 1$, then $\phi(MN) = \phi(M)\phi(N)$. To prove this, write $\gcd(M, N) = 1 = x_0 M + y_0 N$ by Theorem 1.2 and observe that any integer k can be written as $k = kx_0 M + ky_0 N = xM + yN$ for some $x, y \in \mathbf{Z}$. On the other hand, the integers MN and $k = xM + yN$ are relatively prime if and only if $\gcd(x, N) = 1$ and $\gcd(M, y) = 1$. Hence $\{k \in \mathbf{Z} : \gcd(k, MN) = 1\}$ is the set

$$\{xM + yN : \; x, y \in \mathbf{Z}; \; \gcd(x, N) = 1; \; \gcd(M, y) = 1\}.$$

But $xM + yN \equiv x'M + y'N \pmod{MN}$ if and only if $(x - x')M = (y' - y)N + kMN$ for some integer k, which is true if and only if N divides $(x - x')$ and M divides $(y' - y)$, so $xM + yN \equiv x'M + y'N \pmod{MN}$ if and only if $x \equiv x' \pmod{N}$ and $y \equiv y' \pmod{M}$.

Thus each integer in $\{0, 1, \ldots, MN\}$ that is relatively prime with MN is realized as $xM + yN \pmod{MN}$ for exactly one of the $\phi(N)$ representatives $x \in \{0, 1, \ldots, N\}$ and exactly one of the $\phi(M)$ representatives $y \in \{0, 1, \ldots, M\}$ that are relatively prime to N and M, respectively. This implies that $\phi(MN) = \phi(M)\phi(N)$.

Finally, since powers of distinct primes are relatively prime, we can factor ϕ to get the result from the single prime power computation: $\phi(\prod_i p_i^{m_i}) = \prod_i \phi(p_i^{m_i}) = \prod_i (p_i^{m_i} - p_i^{m_i - 1})$. □

Hence the security of RSA depends in part on the complexity of prime factorization.

RSA uses two large primes p, q and puts $M = pq$, so $\phi(M) = (p-1)(q-1)$ by Theorem 1.15. Knowing M and $\phi(M)$ in this case leads to p, q with a little arithmetic, since

$$1 + M - \phi(M) = 1 + pq - (p-1)(q-1) = p + q,$$

and given pq and $p + q$ it is easy to compute $p - q = \pm\sqrt{(p+q)^2 - 4pq}$, from which p and q are individually computable.

Encryption exponent e is chosen[9] relatively prime to $\phi(M)$ and published, but its quasi-inverse $d = e'$ modulo $\phi(M)$, computed with the extended Euler algorithm, must be kept secret.

Decryption of a^e by $a = (a^e)^d \% M$ assumes that $gcd(a, M) = 1$, which is false in the unlikely event that a, which is in some sense random, is divisible by p or q. Further analysis, however, shows that decryption works for all $a \in \{0, 1, \ldots, M-1\}$ in the $M = pq$ case:

Theorem 1.16 (Chinese Remainder) *Suppose $p_1, \ldots, p_m$ are distinct primes and $N \stackrel{\text{def}}{=} p_1 \cdots p_m$. Then for each choice $\{a_1, \ldots, a_m\} \subset \mathbf{Z}$, there is a unique $z \in \{0, 1, \ldots, N-1\}$ satisfying $z \equiv a_i \pmod{p_i}$ for all $i = 1, \ldots, m$.*

[9] It should not be too small; one standard requires $e \geq 65537 = 2^{16} + 1$. This lower bound, incidentally, is a class of prime number called a *Fermat prime*.

Proof: For each $i = 1, \dots, m$, the numbers p_i and N/p_i are relatively prime. Hence by Theorem 1.2 there are integers[10] x_i, y_i satisfying $x_i p_i + y_i(N/p_i) = 1$. Put $b_i \stackrel{\text{def}}{=} y_i N/p_i$ and note that by this construction,

$$b_i \equiv 1 \pmod{p_i}; \qquad b_i \equiv 0 \pmod{p_j}, \quad j \neq i.$$

Now define $z \stackrel{\text{def}}{=} \sum_{i=1}^{m} a_i b_i$ and observe by Lemma 1.8 that $z \equiv a_i \pmod{p_i}$ for all $i = 1, \dots, m$. Hence z is a solution.

For uniqueness, observe that if z and z' are two solutions, then $z - z' \equiv 0 \pmod{p_i}$ for all $i = 1, \dots, m$, so N divides $z - z'$. Hence there is exactly one solution z in $\{0, 1, \dots, N-1\}$. □

Now suppose that $M = pq$ for distinct primes p, q, and $a \in \{0, 1, \dots, M-1\}$ is any integer. Then a is uniquely determined by its remainders $a_p \stackrel{\text{def}}{=} a\%p$ and $a_q \stackrel{\text{def}}{=} a\%q$. Now either $a_p \equiv 0 \pmod{p}$, in which case $a_p^k \equiv 0 \pmod{p}$ for every $k \neq 0$, or else a_p is relatively prime to p and has a quasi-inverse a'_p modulo p that is also relatively prime to p, in which case

$$a_p^{1+n\phi(M)} = a_p^{1+n(p-1)(q-1)} = a_p^{1+nq(p-1)-n(p-1)} = (a_p)^1\,[a_p^{p-1}]^{nq}\,[(a'_p)^{p-1}]^n.$$

By Fermat's Little Theorem the factors in square brackets are congruent to 1 so the expression is congruent to a_p. Conclude that $a_p^{ed} \equiv a_p \pmod{p}$ for any value of a_p if $ed \equiv 1 \pmod{\phi(M)}$. A similar argument shows that $a_q^{ed} \equiv a_q \pmod{q}$. But then $a^{ed} \equiv a \pmod{M}$ is uniquely determined in $\{0, 1, \dots, M\}$ by the Chinese Remainder Theorem, so it will be recovered by the RSA decryption algorithm.

Miller-Rabin primality test

RSA cryptography requires two large primes kept as private data, so they must be chosen in secret for each implementation. One way to do this is to choose a large random integer N and then test it and its successors $N+1, N+2, \dots$ until a prime is found. Trial division is too slow in practice, but there is a faster method based on Fermat's Little Theorem and successive modular square roots:

Lemma 1.17 *If p is a prime and $x^2 \equiv 1 \pmod{p}$, then $x \equiv \pm 1 \pmod{p}$.*

Proof: Factor $(x-1)(x+1) \equiv x^2 - 1 \equiv 0 \pmod{p}$ to conclude that p divides $(x-1)(x+1)$. Since p is prime it divides either $(x-1)$ or $(x+1)$. □

Theorem 1.18 *Suppose N is an odd prime and write $N - 1 = 2^s d$ where d is odd. Then for every integer a with $1 < a < N$, either*

$$a^d \equiv 1 \pmod{N}$$

or

$$a^{2^r d} \equiv -1 \pmod{N}, \qquad \text{for some } 0 \le r \le s-1.$$

[10] These integers may be found by the extended Euclid algorithm.

Proof: By Fermat's Little Theorem, $a^{N-1} \equiv 1 \pmod{N}$. Write

$$1 \equiv a^{N-1} \equiv a^{2^s d} \equiv (a^d)^{2^s} \equiv (\cdots((a^d)^2)^2 \cdots)^2 \pmod{N}.$$

Now use Lemma 1.17 to conclude that one of the square roots is -1 or else all of them down to a^d are $+1$. □

Note that s is easily determined by trial division of N by 2.

A test integer N will be exposed as nonprime if we find a *witness* a such that $a^d \not\equiv 1 \pmod{N}$ and $a^{2^r d} \not\equiv -1 \pmod{N}$ for all $r \in \{0, 1, \ldots, s-1\}$. However, for each nonprime N there are some values of a, called *strong liars*, which behave as if N were prime. It is known[11] that if $N < 341\,550\,071\,728\,321$ then it is enough to test $a \in \{2, 3, 5, 7, 11, 13, 17\}$, for not all of these can be strong liars.

It is believed that for any N the set $\{2, 3, \ldots, \lfloor 2(\ln N)^2 \rfloor\}$ must contain a witness so that checking this set settles the question of N's primality, but that result follows from the Riemann Hypothesis which no one currently knows how to prove.

1.1.2 Representing integers in binary computers

Computers have *internal representations* for numbers that in most cases are easily translated to binary, or base-2, notation. Most humans, on the other hand, use decimal or base-10 notation. Binary notation in this text will be indicated by a string of binary digits, or *bits*, each taking the value 0 or 1, followed by "base 2" in parentheses. A four-bit binary number will look like $b_3b_2b_1b_0$ (base 2); one specific example is the number 9, which is 1001 (base 2). This is analogous to a four-digit decimal number like $d_3d_2d_1d_0$ (base 10), for example 1997 (base 10). The "base 10" is usually omitted.

The base can be any positive integer greater than one.[12] A number can be written in any base, and its value for use in arithmetic can be obtained by summing the values represented by its digits:

$$\begin{aligned}
1001 \text{ (base 2)} &= 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 9 \text{ (base 10)}; \\
1997 \text{ (base 10)} &= 1 \times 10^3 + 9 \times 10^2 + 9 \times 10^1 + 7 \times 10^0 \\
&= 1024 + 512 + 256 + 128 + 64 + 8 + 4 + 1 \\
&= 2^{10} + 2^9 + 2^8 + 2^7 + 2^6 + 2^3 + 2^2 + 2^0 \\
&= 11111001101 \text{ (base 2)}.
\end{aligned}$$

More generally, the n-digit number written as $h_{n-1} \ldots h_1 h_0$ (base B) has the value $h_{n-1} \times B^{n-1} + \cdots + h_1 \times B^1 + h_0 \times B^0$. The digits $h_0, h_1, \ldots$ must lie in the range $\{0, 1, \ldots, B-1\}$. The choice $B = 16$ gives the useful *hexadecimal* system which uses the digits $\{0, \ldots, 9, A, B, C, D, E, F\}$, where $A = 10$, $B = 11$, $C = 12$, $D = 13$,

[11] See Jaeschke, G., *On Strong Pseudoprimes to Several Bases.* Mathematics of Computation volume 61, pages 915-926, published in 1993.

[12] We will not consider tallies, like $3 = |||$ or $7 = |||||||$, that give "base one" notation.

Hex	Bin	Hex	Bin	Hex	Bin	Hex	Bin
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

Table 1.1: One hexadecimal digit corresponds to four binary digits, or bits.

$E = 14$, and $F = 15$. Hexadecimal and binary are related by Table 1.1. Each hexadecimal digit corresponds to 4 bits, making it easy to find the corresponding binary expression: 1997 (base 10) = $7CD$ (base 16) = 0111 1100 1101 (base 2).

The binary digits $\{b_i : i = 0, 1, \ldots, n-1\}$ of an n-bit nonnegative integer $x = b_{n-1} \cdots b_2 b_1 b_0$ (base 2) are printed by the following algorithm:

Print the Binary Digits of an Integer $x \geq 0$, LSB First

```
bits( x ):
[0] Print x%2
[1] If x>0, then call bits(x/2)
[2] Return
```

Notice that `bits(x)` prints the *least significant bit* (LSB) first, which is always b_0, then b_1, b_2, and so on, terminating with the *most significant bit* (MSB), the leftmost "1" of x as it is usually written in binary. If $x = 0$ this routine prints a single 0. In particular, `bits(x)` may print fewer than n bits.

To obtain the most significant bit first, merely interchange the print and recursive function call lines:

Print the Binary Digits of an Integer $x \geq 0$, MSB First

```
bits0( x ):
[0] If x>0, then call bits0(x/2)
[1] Print x%2
[2] Return
```

Notice that this function always prints a single leading 0 followed by the most significant bit, then the rest down to bit b_0. Thus `bits(x)` may print up to $n+1$ bits or it may print fewer than n bits.

The function `bits()` may also be implemented nonrecursively:

Print the Binary Digits of an Integer $x \geq 0$, LSB First, Nonrecursively

```
bitsnr( x ):
[0] Print x%2
[1] Let x = x/2
[2] If x>0, then go to [0]
```

Modular exponentials like $a^e \% M$ are efficiently computed using the binary ex-

pansion $e = e_n 2^n + \cdots + e_1 2 + e_0 = \sum_{k=0}^{n} e_k 2^k$:

$$a^e = (a)^{e_0} \times (a^2)^{e_1} \times \cdots \times (a^{2^n})^{e_n} = \prod_{k=0}^{n} (a^{2^k})^{e_k} = \prod_{\{k:e_k=1\}} (a^{2^k}),$$

since the factors with $e_k = 0$ are all 1. The powers a^{2^k} are obtained recursively by squaring:

$$a^{2^0} = a; \qquad a^{2^k} = (a^{2^{k-1}})^2, \quad k = 1, 2, \ldots$$

Likewise, the modular powers $a^{2^k}\%M$ are obtained by squaring and then finding the remainder modulo M. The function `bitsnr()` may be modified to perform this exponentiation:

Compute a^e Modulo M Using the Binary Digits of e

```
modular_power( a, e, M ):
[0] Let prod = 1
[1] If e%2 == 1 then let prod = (a*prod) % M
[2] Let a = (a*a) % M
[3] Let e = e/2
[4] If e>0, then go to [1]
[5] Print prod
```

There are $\log_2 e$ passes through the loop `[1]–[4]`, each of which costs a fixed number of multiplications, so the total cost is $O(\log_2 e)$ rather than the $O(e)$ needed for the naïve algorithm.

The digits in base B of a nonnegative integer x are printed by another generalization of `bits()`. Recall that, if $x \geq 0$ and $B > 0$ are integers, then $x\%B$ is the remainder left after dividing x by B.

Print the Base-B Digits of an Integer $x \geq 0$

```
digits( x, B ):
[0] Print x%B
[1] If x>0, then call digits(x/B,B)
[2] Return
```

Like `bits()`, this function prints the least significant digit first. It has a nonrecursive version as well as a recursive version that prints the most significant bit first. Of course, if $B > 10$, then suitable letters could be printed to represent digit values from 10 to $B - 1$.

This recursive function is easily modified to print the most significant digits first. There is also a nonrecursive version analogous to `bitsnr()`. Both are left as exercises.

1.1.3 Integer arithmetic

A binary computer that stores w bits per integer has a maximum unsigned integer of $2^w - 1$. In general, a computer that stores integers as w base-B digits has a maximum unsigned integer of $B^w - 1$. However, the case $B \neq 2$ is rare enough to be ignored hereafter. Some common values for w are 8, 16, 24, 32, 36, 64, 80, 96, or 128, typically with a selection of several being available. For example, a program written in the Standard C language on one machine *host* can use integer variables of type `char` ($w = 8$), `short` ($w = 16$), `int` ($w = 32$), or `long` ($w = 64$). These parameters are stored in a file named `limits.h` on each machine:

Excerpt from `limits.h`

```
#define    CHAR_BIT                        8
#define    SCHAR_MIN                    -128
#define    SCHAR_MAX                     127
#define    UCHAR_MAX                     255
#define    SHRT_MIN                   -32768
#define    SHRT_MAX                    32767
#define    USHRT_MAX                   65535
#define    INT_MIN               -2147483648
#define    INT_MAX                2147483647
#define    UINT_MAX               4294967295
#define    LONG_MIN     -9223372036854775808
#define    LONG_MAX      9223372036854775807
#define    ULONG_MAX    18446744073709551615
```

In particular, 32-bit binary integers of type `unsigned int` can take one of the $2^{32} = 4\,294\,967\,296$ possible values between 0 and $2^{32} - 1 = 4\,294\,967\,295$. The special name `UINT_MAX` is given to this *maximum unsigned integer*, or largest counting number. If x and y are positive integers with $x + y <$ `UINT_MAX`, then the addition $x + y$ will be correctly computed if performed using variables of type `unsigned int`. Otherwise, the result will be the unique $z \in \{0, 1, \ldots,$`UINT_MAX`$\}$ that satisfies $x + y \equiv z$ (mod `UINT_MAX`+1).

For positive x, y with $x + y >$ `UINT_MAX`, it may be possible to perform ordinary addition with variables of greater bit width such as `long int` or `unsigned long int`. The 1999 C standard includes the types `long long int` and `unsigned long long int`, too, which may have even more bits on a particular host.

There are two common ways of representing negative integers at fixed binary precision. The first is called *sign and magnitude form*; it consists of treating the most significant bit as a sign bit. If 0, the number is positive. If 1, the number is negative. The remaining bits are taken to be the absolute value of the integer and are interpreted as counting numbers. This method wastes one binary string representing -0. To change $x \mapsto -x$, just change the sign bit to its complement. The most negative 32-bit integer representable by sign and magnitude is $-(2^{31} - 1) = -2\,147\,483\,647$, and the largest positive signed integer is $2^{31} - 1 = 2\,147\,483\,647$ in 32-bit ones-

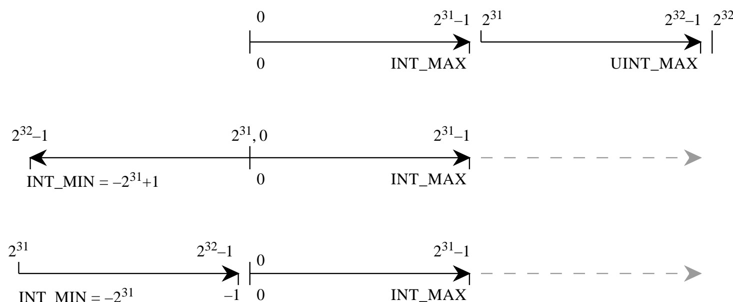

Figure 1.1: Top: Unsigned 32-bit integers. Middle: Sign and magnitude interpretation. Bottom: Twos complement interpretation. Numbers above each line are the counting values, while those below the line are the represented values.

complement arithmetic. Standard C defines `INT_MIN` and `INT_MAX`, respectively, to have these values on each host computer employing sign and magnitude form.

The second, more common integer representation is called *twos complement form*. In this method, using w bits for a total of $2^w - 1$ numbers, the low half $[0, 2^{w-1} - 1]$ represent nonnegative integers, while the high half $[2^{w-1}, 2^w - 1]$ represent negative integers to which 2^w has been added. The most significant bit again determines the sign: 1 means negative, 0 means positive. This arrangement is depicted in Figure 1.1. It in turn has the drawback that the negative of a representable integer is not always representable since `-INT_MIN` is larger than `INT_MAX`.

The twos complement form of $-x$ for a w-bit integer x is the counting number represented by $2^w - x$, that is, the additive inverse of x modulo 2^w. It is therefore a simple bitwise operation to compute $x \mapsto -x$ in twos complement form: first find the *ones-complement* by flipping $0 \leftrightarrow 1$ all the bits of x, and then increment $x \leftarrow x+1$. Flipping, or *complementing* a w-bit number x is the same as subtracting it from $2^w - 1$, which is a string of w 1-bits, so these operations give $[(2^w - 1) - x] + 1 = 2^w - x$. For example, with $w = 8$ and $x = 13 = 00001101$ (base 2), the ones complement of x is $242 = 11110010$ (base 2), and the twos complement is $243 = 11110011$ (base 2), which is congruent to $-13 \pmod{256}$.

Twos complement w-bit integer arithmetic is implemented in hardware as arithmetic modulo 2^w, with addition, subtraction, multiplication, integer division and remainder performed by dedicated circuitry. Some checks must be built in, though. It is possible to add two positive integers and get the representation of a negative integer, for example $100 + 99$ gives the 8-bit twos complement representation for -57. This is called an *integer overflow*. Likewise, $-100 - 99$ produces the *integer underflow* value 57. Underflow or overflow occurs if and only if the carry into the sign or most significant bit is different from the carry out of the sign bit.

Logical operations such as order comparison and equality testing can be implemented by subtraction modulo 2^w followed by testing the sign bit or testing if all bits are zero.

1.2 Real Numbers

Denote by $\mathbf{Q}$ the set of *rational numbers* p/q, each of which is described[13] by a numerator $p \in \mathbf{Z}$ and a denominator $q \in \mathbf{Z}$, $q \neq 0$. Each element of $\mathbf{Q}$ is therefore completely described by a finite list of symbols. Of course, $p/q = p'/q'$ if and only if $pq' = p'q$, but we can always find the unique representative of p/q in *lowest terms* by the reduction $p \leftarrow p/\gcd(p,q)$, $q \leftarrow q/\gcd(p,q)$, followed by changing the sign of both p and q, if necessary, to make $q > 0$. A computer can store one rational number in the same space needed for one integer, just by assigning a fixed subset of bits for a nonnegative integer numerator, one bit for the sign, and the remaining bits for the positive integer denominator.

Devices to perform rational number arithmetic are combinations of devices that perform integer arithmetic. For example, $p/q+p'/q' = (pq'+p'q)/qq'$ requires three integer multiplications and one addition. The sign is carried by the numerator while the denominators remains positive. Likewise, comparisons are derived from integer comparisons: $p/q < p'/q'$ if and only if $pq' < p'q$, and so on.

Fractions and their "decimal expansions" can be written in any base just like integers, using the following interpretation: evaluate $x = h_{n-1} \dots h_1 h_0 . h_{-1} h_{-2} \dots h_{-m}$ (base B) as

$$x = h_{n-1} \times B^{n-1} + \cdots + h_1 \times B + h_0 + \frac{h_{-1}}{B} + \frac{h_{-2}}{B^2} + \cdots \frac{h_{-m}}{B^m}. \tag{1.3}$$

The "decimal" point separates the integer and fractional parts of the number. In this example, the fractional part of x can be rewritten as

$$\frac{h_{-1} \times B^{m-1} + h_{-2} \times B^{m-2} + \cdots + h_{-m}}{B^m}. \tag{1.4}$$

The integer numerator from Equation 1.4, which is B^m times the fractional part of x, may be used as input to the base-B conversion program. Its output will be the m digits to the right of the decimal point printed least-significant-digit first. Then a decimal point may be printed, followed by the digits of the integer part printed least-significant-digit first:

Print the Integer Part and the First m Base-B Digits of $x \geq 0$

```
places( x, m, B ):
[0] Let xf = fractional part of x, let xi = integer part of x
[1] For i = 1 to m, replace xf *= B
[2] For i = 1 to m do [3] to [4]
[3]     Print xf%B
[4]     Let xf = xf/B
[5] Print a decimal point
[6] Call digits(xi, B)
```

[13] We write p/q for convenience, we do not actually divide.

Steps 2–4 of `places()` must be implemented differently than `digits()` so as to print the leading zeros that the fractional part sometimes needs as decimal placeholders. The same idea may be used to implement `digits()` nonrecursively, but that is left as an exercise.

It has been known since ancient times that fairly simple problems have no exact solutions in the rational numbers. A famous example, due to Euclid, is that $\sqrt{2}$ cannot be expressed as p/q for integers p, q since then $p^2/q^2 = 2$ implies that p^2 is even, so p must be even, and then p^2 is really divisible by 4 so q^2 and thus q must be even. Hence p/q is not in lowest terms. But this applies to every $p/q = \sqrt{2}$, so were $\sqrt{2}$ rational, it would have no representative in lowest terms, which cannot be. Hence we cannot solve the problem $x^2 = 2$ with $x \in \mathbf{Q}$.

We can easily find approximate solutions $x \in \mathbf{Q}$ to the problem $x^2 = 2$:

$$\begin{aligned}
1^2 &\le 2 \le 2^2 &&\Rightarrow 1 \le \sqrt{2} \le 2;\\
1.4^2 &\le 2 \le 1.5^2 &&\Rightarrow 1.4 \le \sqrt{2} \le 1.5;\\
1.41^2 &\le 2 \le 1.42^2 &&\Rightarrow 1.41 \le \sqrt{2} \le 1.42;\\
1.414^2 &\le 2 \le 1.415^2 &&\Rightarrow 1.414 \le \sqrt{2} \le 1.415;\\
&\ \ \vdots\\
1.4142135623^2 &\le 2 \le 1.4142135624^2\\
&\ \ \vdots &&\Rightarrow 1.4142135623 \le \sqrt{2} \le 1.4142135624;
\end{aligned}$$

This procedure can be continued indefinitely with each step taking a fixed finite amount of work to shrink the difference between the upper and lower estimate by a factor of 10. It produces a *Cauchy sequence*, an unending list $\{x_1, x_2, \ldots\}$ of numbers with the property that for every positive integer d, there is some starting index $N = N(10^{-d})$ such that $x_N, x_{N+1}, \ldots$ all have the same first d digits in their decimal expansion. The rational approximations to $\sqrt{2}$ given by the lower bounds $\{1, 1.4, 1.41, 1.414, \ldots\}$ define a Cauchy sequence with $N(10^{-d}) = d$ since the first d digits of x_n will be the same for all $n \ge d$. It is an easy exercise to show that the following more traditional definition is equivalent:

Definition 2 *The list* $\{x_1, x_2, \ldots\}$ *is a* Cauchy sequence *if, for every* $\epsilon > 0$*, there is some* $N = N(\epsilon)$ *such that* $|x_n - x_m| < \epsilon$ *whenever both* $n \ge N$ *and* $m \ge N$.

We may define the *real numbers* $\mathbf{R}$ to be the enlargement of $\mathbf{Q}$ that includes all the *infinite decimal expansions* obtained using Cauchy sequences of rational numbers. Since rational numbers themselves have decimal expansions, this means that $\mathbf{R}$ is the set of infinite decimal expansions. Numbers like 1 can also be considered infinite decimals $1.00\ldots$, and so on.

Not only are there infinitely many real numbers, but most of them cannot be specified exactly since that would require writing infinitely many digits. However, a fixed-precision approximation to a real number often suffices. Humans write such approximate values in *scientific notation* using base 10. For example, we write $.314159 \times 10^1$ for the six-digit approximation to π. Computers unable to print

superscripts would write `.314159e+01`. The *mantissa* or *fractional part* `314159` contains the six digits, while the *exponent* `+01` indicates how to move the decimal point, here one digit to the right, in order to get the usual decimal expansion 3.14159.

Note that $1 = 0.999\ldots$, since $x = 0.99\ldots$ satisfies $10x - x = 9$. Thus two different decimal expansions can represent the same real number. We will say that two numbers $x, y \in \mathbf{R}$ are equal if, for any Cauchy sequences $\{x_n\}$ and $\{y_n\}$ representing x and y, respectively, the Cauchy sequence $\{x_n - y_n\}$ represents 0. That applies to $\{x_n\} = \{1, 1, 1, \ldots\}$ and $\{y_n\} = \{0.9, 0.99, 0.999, \ldots\}$ since $\{x_n - y_n\} = \{0.1, 0.01, 0.001, \ldots\}$ which evidently represents 0.

The real number x represented by a Cauchy sequence $\{x_n : n = 1, 2, \ldots\}$ is called its *limit*, and we write $x = \lim_{n\to\infty} x_n$. For every nonnegative integer d, there is an integer $N = N(10^{-d})$ such that the first d digits of $x_N, x_{N+1}, \ldots$ all agree with the first d digits of x. This is equivalent to the traditional definition:

Definition 3 *We say that the limit $x = \lim_{n\to\infty} x_n$ exists if, for every $\epsilon > 0$, there is some $N = N(\epsilon)$ such that $|x - x_n| < \epsilon$ whenever $n \geq N$.*

If the limit of some sequence $\{x_n\}$ exists, we also say that x_n *converges to x as n tends to infinity*, and write $x_n \to x$ as $n \to \infty$. By our construction, every Cauchy sequence of rational numbers has a unique real number as a limit. But once we include those limits, we have a complete set:

Theorem 1.19 (Completeness of R) *A Cauchy sequence of real numbers has a unique real-number limit.*

Proof: Suppose $x_1, x_2, \ldots$ is a Cauchy sequence of real numbers. For each $d = 1, 2, \ldots$, let $N = N(10^{-d})$ be an integer such that the first d digits of $x_N, x_{N+1}, \ldots$ are the same. The first d digits of x_N define a rational number[14], which we may call x'_d. But then $\{x'_1, x'_2, \ldots\}$ is a Cauchy sequence of rational numbers that converges to a unique real number, which we may denote by x. But x_n converges to x too, since for every $\epsilon > 0$ there is some d with $10^{-d} < \epsilon$, and $|x_n - x| < \epsilon$ for all $n > N(10^{-d})$. □

1.2.1 Precision and accuracy

Accuracy is the difference between the exact value of a quantity and its approximate value. *Precision* is the difference between two adjacent approximate values. To say that π is approximately 3.140000000 is to be precise but inaccurate. Using great precision for quantities known to low accuracy is misleading.

Suppose that $x_0 \neq 0$ is a real number, considered to be the exact value of a quantity, and x is another real number used to approximate x_0. For example, x might be an integer, or a fraction with denominator 2^m representing m bits after the decimal point such as the output of `places(x0,m,2)` defined on page 19.

[14]What power of 10 will be the denominator?

Definition 4 *Put* $\Delta x \stackrel{\text{def}}{=} x - x_0$. *Then:*

- *The* absolute error *of approximating* x_0 *by* x *is* $|x - x_0| = |\Delta x|$;
- *The* relative error *of approximating* x_0 *by* x *is* $|x - x_0|/|x_0| = |\Delta x|/|x_0|$;
- *The number of* digits of accuracy *in* x *is the largest natural number* d *such that* $10^d|\Delta x|/|x_0| \leq 5$. *If the relative error is smaller than 1, then*

$$d = \left\lfloor \log_{10}\left(\frac{5|x_0|}{|\Delta x|}\right)\right\rfloor.$$

- *The number of* bits of accuracy *in an approximation of* $x_0 \neq 0$ *is the largest natural number* b *such that* $2^b|\Delta x|/|x_0| \leq 1$. *Again, if the relative error is smaller than 1, then*

$$b = \left\lfloor \log_{2}\left(\frac{|x_0|}{|\Delta x|}\right)\right\rfloor.$$

Bits and digits of accuracy are related; each digit of accuracy is worth $\log_2(10) \approx 3.32$ bits, so $b \approx 3.32d$.

When $x_0 = 0$, absolute error is still $|\Delta x|$, but relative error is undefined and digits of accuracy is calculated using absolute error.

To say that 3.14159 is the six-digit approximation to π means that $3.14158 < \pi < 3.14160$. The absolute error $|3.14159 - \pi|$ is also called the *round-off error*, or sometimes the *truncation error*. It is always smaller than one unit at the least significant digit of the mantissa. The *error interval* is determined by how the approximate value was chosen. It is $3.14159 \leq \pi < 3.14160$ if the approximation just truncated a longer decimal expansion, but it is $3.141585 \leq \pi \leq 3.141595$ if the approximation rounded to the nearest six-digit decimal expansion.

1.2.2 Representing real numbers

A computer can only distinguish among finitely many *representable values*, lying in some bounded range, and the format in which this is usually done is called *floating-point*. Replacing a real number with a representable value introduces round-off error which is relatively small if the real number lies in the bounded range. However, there are always real numbers much larger and much smaller than any representable value, which cannot be approximated with small round-off error. It is common to treat such values as $\pm\infty$ and devise special arithmetic rules for them.

Binary computers keep *internal representations* of floating-point numbers as a string of binary digits, just like integers, but the bits are interpreted differently. A fixed number of bits give the sign and the base-two digits of the mantissa, while the rest give the sign and magnitude of the exponent, just as in scientific notation. Arithmetic algorithms for such strings of bits should be simple, so that logic circuits to compute sums, products, sign changes, comparisons and so on are as simple as possible. Unfortunately, there are many reasonable solutions to this design problem,

so that different computers might use different floating-point formats and different sets of representable values. They might therefore produce different outputs even when running the same algorithm on identical inputs.

It is nevertheless possible to impose standards that control the maximum difference between the outputs of an algorithm on different machines. For example, the Standard C programming language requires each host computer to have a standard header file `float.h` that lists the number of bits devoted to the mantissa as well as the smallest and largest representable positive numbers in two common formats: float or single precision, and double precision.

Excerpt from `float.h`

```
#define     FLT_RADIX                    2
#define     FLT_ROUNDS                   1
#define     FLT_MANT_DIG                24
#define     FLT_MIN_EXP               -125
#define     FLT_MAX_EXP               +128
#define     FLT_MIN         1.17549435e-38
#define     FLT_MAX         3.40282347e+38
#define     FLT_EPSILON     1.19209290e-07
```

`FLT_RADIX` is the base of the number system, here 2, for binary. `FLT_ROUNDS` indicates how the machine chooses representations for real numbers:

`-1`: no rounding is specified.

`0`: round toward 0. Choose the nearest representable value whose absolute value is no greater than the number.

`1`: round toward the nearest representable value. Ties are broken with a convention, for example, always choosing the representable value whose least significant mantissa digit is even. Such a rule makes the expected round-off error zero.

`2`: round toward $+\infty$, that is, round up. Choose the nearest representable value greater than or equal to the number.

`3`: round toward $-\infty$, that is, round down. Choose the nearest representable value less than or equal to the number.

The next three numbers specify how many digits there are in the mantissa, `FLT_MANT_DIG`, and the minimum `FLT_MIN_EXP` and maximum `FLT_MAX_EXP` values for the exponent.

Standard C also specifies `FLT_MIN` and `FLT_MAX`, respectively the minimum and maximum *normalized* positive numbers representable by the computer, namely those that have a nonzero first digit in the mantissa. Smaller positive numbers are representable by using the minimum exponent and mantissas beginning with zero, but they fall into a special class called *subnormal* numbers and have fewer digits of precision.

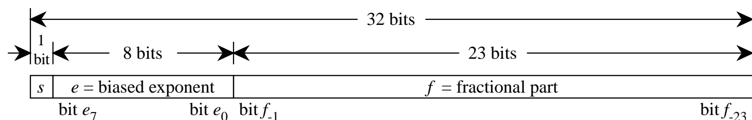

Figure 1.2: Schematic arrangement of bit fields in the single-precision (32-bit) IEEE binary floating-point format.

The *floating-point epsilon*, `FLT_EPSILON`, is a key measure of precision and truncation error. It is the difference between the floating-point representation of 1, which is exact, and that of the "next larger" representable floating-point number. On the example binary computer, this is $2^{-23} \approx 1.19209 \times 10^{-7}$, the value of the least significant bit in a 24-bit mantissa divided by the value of the most significant bit. This bounds the relative error of truncating a real number to the number of digits available to the computer. It shall be called ϵ_f in this text. A related quantity is the least positive number ϵ_0 such that the floating-point representation of $1 + \epsilon_0$ is different from that of 1. That is, the computer evaluates the comparison $1.0 + \epsilon_0 > 1.0$ as true, but considers $1.0 + \epsilon = 1.0$ for any positive ϵ less than ϵ_0. Any real number can be represented by the computer with a relative error less than ϵ_0. In all cases, $0 < \epsilon_0 \leq \epsilon_f$; on machines that round to the nearest representable value, $\epsilon_0 = \frac{1}{2}\epsilon_f$.

With this information, it is possible to compute the maximum error in a particular computation from particular inputs. Two implementations of an algorithm may be considered equivalent if, for sufficiently many and varied inputs, their outputs differ by no more than the sum of those maximum errors.

IEEE floating-point formats

The Institute for Electrical and Electronics Engineers, or IEEE, sponsored a committee that in 1985 published a standard format for 32-bit binary floating-point computer arithmetic. The standard effectively defines a function $v : \mathbf{R} \to \mathbf{R}$ mapping any real number x to its nearest representable value $v(x)$. It does that by specifying how to store representable values as bit strings. Figure 1.2 shows how bits are allocated into three fields s, e, and f in that format.

The s bit is 0 for positive numbers and 1 for negative numbers, that is, the sign of the represented value is that of $(-1)^s$. It is followed by an 8-bit exponent field in which an unsigned integer is stored, most-significant bit first, as $e = e_7 \cdots e_1 e_0$ (base 2). This e is called the *biased exponent*, and it takes the values 0,1,...,255. To obtain a signed value, a *bias* of 127 is subtracted to get the *unbiased exponent:*

$$E = e - 127 = e_7 \cdot 2^7 + \cdots + e_1 \cdot 2 + e_0 - 127. \tag{1.5}$$

Thus $-127 \leq E \leq +128$. However, the values $E = -127 \leftrightarrow e = 0$, as well as $E = +128 \leftrightarrow e = 255$, are reserved to indicate special numbers like $\pm\infty$, so the valid range of unbiased exponents in this format is $-126 = E_{min} \leq E \leq E_{max} = +127$.

Let v be the real number represented by the bit strings s, e, f. Then v has the following interpretation:

- If $e = 255$ and $f \neq 0$, then v is *Not a Number (NaN)* regardless of the value of s. This value is considered different from $\pm\infty$ and can be used to signal invalid results.
- If $e = 255$ and $f = 0$, then $v = (-1)^s \infty$.
- If $0 < e < 255$, then $v = (-1)^s \, 2^{e-127} \, (1.f_{-1} \cdots f_{-23} \text{ (base 2)})$. This is also written $v = (-1)^s \, 2^E \, (1.f)$. Normalized mantissas must have nonzero first digit, so the fractional part of the number is supplied with a leading one: $1.f = 1 + f_{-1} \cdot 2^{-1} + \cdots + f_{-23} \cdot 2^{-23}$. This gives a precision of 24 bits while using only 23 actual bits. The extra bit of information is deduced from the exponent.
- If $e = 0$ and $f \neq 0$, then $v = (-1)^s \, 2^{-126} \, (0.f_{-1} \cdots f_{-23} \text{ (base 2)})$ is a *subnormal* or tiny number. Subnormal mantissas have leading digit zero: $0.f = f_{-1} \cdot 2^{-1} + \cdots + f_{-23} \cdot 2^{-23}$.
- If $e = 0$ and $f = 0$, then $v = (-1)^s \, 0$. For some operations, $+0$ and -0 are distinguished. In particular, $\sqrt{-0} = -0$ while $\sqrt{+0} = +0$. However, the comparison $-0 == +0$ evaluates as true on all machines conforming to this standard.

There is also a *double precision* format in which the unbiased exponent contains 11 bits, the normal bias is 1023, the subnormal bias is 1022, there are 52 actual mantissa bits for a normal precision of 53 bits, and there is one sign bit. Conversion from single to double precision is exact, but converting from double to single precision involves rounding.

The floating-point format specifies certain arithmetic operations that must be available such as addition, division, absolute value, and extraction of square roots. It further requires that the output of those operations be the representable value obtained by first performing the calculation in exact arithmetic and then rounding to the available precision. This is achieved in practice by storing intermediate results of calculations in *extended* formats at higher precision than the final result. The IEEE requires extended formats to have at least a certain width, and also that either each representable number is encoded as a unique bit string, or else different bit strings representing the same value are not distinguished in any operation. For example, the *extended double precision* format must have at least 64 bits of precision and at least 79 bits total.

Computers with dedicated floating-point arithmetic circuitry typically perform all floating-point calculations in the extended double format, then round for output into either the single or double precision storage formats. The IEEE requirements ensure that any peculiarities of the implementation remain invisible to the user, and guarantee that all computers conforming to the standard will produce identical results given identical inputs.

Conversion from decimal notation into the binary format are also treated by the standard, since many programs contain parameters entered by humans using scientific notation. Conversion is required to be *monotonic*, that is, if $x \geq y$ are two real numbers represented in scientific notation, then the associated representable value $v(x)$ must not be smaller than $v(y)$. All of the rounding procedures described in the previous section are monotonic.

1.2.3 Propagation of error

In IEEE floating-point format with maximum representable value $M =$ `FLT_MAX` and minimum normalized positive representable value $m =$ `FLT_MIN`, the *normally representable set* is defined by $NR \stackrel{\text{def}}{=} [-M,-m] \cup [m,M]$. It consists of the nonzero real numbers x that are approximated by $v(x)$ within relative error ϵ_f:

$$\left|\frac{v(x)}{x} - 1\right| \leq \epsilon_f \iff 1 - \epsilon_f \leq \frac{v(x)}{x} \leq 1 + \epsilon_f$$

Typically, $0 < \epsilon_f < 10^{-6} \ll 1$, so this implies that $v(x) \neq 0$ and that the reciprocal $x/v(x)$ satisfies essentially the same bounds:

$$1 - \epsilon_f + O(\epsilon_f^2) = \frac{1}{1+\epsilon_f} \leq \frac{x}{v(x)} \leq \frac{1}{1-\epsilon_f} = 1 + \epsilon_f + O(\epsilon_f^2)$$

The higher order terms may be ignored for typical ϵ_f, or we may preserve exact inequality simply by replacing $\epsilon_f \leftarrow \epsilon_f/(1-\epsilon_f)$. Alternatively, in a computer that rounds to the nearest representable value as indicated by `FLT_ROUNDS=1`, the relative error bounds hold with $\epsilon_0 = \frac{1}{2}\epsilon_f$, making both inequalities exact without any modification of ϵ_f. Then we may multiply either expression by the denominator to get equivalent bounds for $\Delta x \stackrel{\text{def}}{=} x - v(x)$, the *error of representation*:

$$|\Delta x| \leq \epsilon_f \, |v(x)|; \qquad |\Delta x| \leq \epsilon_f \, |x| \tag{1.6}$$

Conditioning of arithmetic

A computation $x \mapsto y$ is said to be *well-conditioned* if the relative error in y is comparable to the relative error in x. On a finite precision computer, the input x can have a relative error as large as ϵ_f, so the conditioning of a single calculation "from exact inputs" is usually stated as the multiple of ϵ_f that we get for the relative error of y, given a relative error of ϵ_f in x.

When there are several inputs $(x_1, \ldots, x_n) \mapsto y$, it is assumed that they all have relative error ϵ_f, with the "worst" combination of signs.

An ill-conditioned calculation is one that can greatly magnify relative error. Ill conditioning arises in truncated infinite algorithms if many steps combine to greatly magnify small initial errors. For example, the Fibonacci sequence of Equation 1.1 begins with the initial values $F_0 = 0$ and $F_1 = 1$. Starting with different initial values $F_0' = 1$ and $F_1' = \varphi_-$ gives a sequence $F_k' = \varphi_-^k \to 0$ as $k \to \infty$, since

$|\varphi_-| < 1$. However, modifying the different initial condition however slightly to $F'_0 = 1 + \epsilon$ for any $\epsilon > 0$ gives a sequence $|F'_k| \to \infty$ as $k \to \infty$. By choosing large enough k we can magnify the relative error in F'_k, compared to the relative error ϵ in F'_0, by an arbitrarily large amount.

Sums and differences can be ill-conditioned because of *catastrophic cancellation.* Suppose x, y, $x + y$, and $v(x) + v(y)$ all belong to NR. The IEEE procedure to compute $x + y$ is first to approximate x by $v(x)$ and y by $v(y)$, then to find $v(x) + v(y)$ in exact arithmetic, and finally to find the nearest representable value $v\,(v(x) + v(y))$. The absolute error after this calculation is the difference between the exact value $x+y$ and the computed value $v\,(v(x) + v(y))$. It can be estimated as follows, regardless of the rounding method used to obtain the representable value:

$$\begin{aligned} |x + y - v\,(v(x) + v(y))\,| &\leq |x - v(x)| + |y - v(y)| \\ &\quad + |v(x) + v(y) - v\,(v(x) + v(y))\,| \\ &\leq (|x| + |y| + |v(x) + v(y)|)\,\epsilon_f. \end{aligned}$$

Dividing by $|x+y|$ and ignoring the $O(\epsilon_f^2)$ terms gives the relative error of the sum:

$$\left|\frac{x + y - v(v(x) + v(y))}{x + y}\right| \leq \left(1 + \frac{|x| + |y|}{|x + y|}\right)\epsilon_f. \tag{1.7}$$

The three terms contributing to this error can differ greatly in magnitude. For example, if $x + y \approx 0$ but $x \approx -y \approx 1$, then $(|x| + |y|)/|x + y|$ is much greater that 1 and thus dominates the error estimate. In this case it is possible that the relative error in $x + y$ vastly exceeds the relative error in x or y. As an example, consider approximating $f(t+h) - f(t)$ with $f(t) = t^2$, $t = 1.112233$, and $h = 1.000000 \times 10^{-6}$ using single-precision arithmetic. Then $f(t + h) = 1.112234^2 \approx 1.237064$ and $f(t) = 1.112233^2 \approx 1.237062$, so

$$f(t + h) - f(t) = 1.237064 - 1.237062 = 0.000002 = 2.000000 \times 10^{-6},$$

instead of the correctly rounded single-precision value 2.224467×10^{-6} obtained from exact arithmetic. The relative error in the result is around 1.1×10^{-1} instead of $\epsilon_f \approx 1.2 \times 10^{-7}$; it has been magnified about a millionfold.

However, there is no cancellation when x and y have the same sign, so $|x| + |y| = |x + y|$, making the relative error of the sum no more than $2\epsilon_f$.

Products and quotients, by contrast, are always well-conditioned. If x, y, xy, and $v(x)v(y)$ all belong to NR, then an argument like that used to prove the product rule in calculus shows:

$$\begin{aligned} |xy - v(v(x)v(y))| &\leq |\,(x - v(x))\,y| + |\,(y - v(y))\,v(x)| \\ &\quad + |v(x)v(y) - v(v(x)v(y))| \\ &< 3\epsilon_f\,|xy|. \end{aligned}$$

The final step depends on Equation 1.6 and the assumption that $0 < \epsilon_f \ll 1$. Hence,

$$\left|\frac{xy - v(v(x)v(y))}{xy}\right| < 3\epsilon_f, \tag{1.8}$$

so the relative error in the product xy is no more than three times the maximum relative error ϵ_f of each of the factors.

Quotients are products involving a reciprocal. The error of calculation for a reciprocal is

$$\left|\frac{1}{x} - v\left(\frac{1}{v(x)}\right)\right| \le \left|\frac{1}{x} + v\left(\frac{1}{x}\right)\right| + \left|v\left(\frac{1}{x}\right) - v\left(\frac{1}{v(x)}\right)\right| < 2\epsilon_f \left|\frac{1}{x}\right|.$$

Here it is assumed that $x \neq 0$, $x \in NR$, and $1/v(x) \in NR$, as well as that 1 is exactly representable: $v(1) = 1$. The final step uses Equation 1.6 under the assumption that $0 < \epsilon_f \ll 1$. Hence the relative error of calculating the reciprocal is

$$\left|\frac{\frac{1}{x} - v\left(\frac{1}{v(x)}\right)}{\frac{1}{x}}\right| < 2\epsilon_f,$$

so each reciprocal in a product adds at most $2\epsilon_f$, rather than at most ϵ_f, to the relative error of the result. Combining the product and reciprocal formulas gives

$$\left|\frac{\frac{y}{x} - v\left(\frac{v(y)}{v(x)}\right)}{\frac{y}{x}}\right| < 4\epsilon_f. \tag{1.9}$$

Thus quotients, like products, are well-conditioned computations.

*Functions

Suppose that $F = F(x)$ is a function of one real variable that satisfies a *Lipschitz condition*, namely that it has some constant $M = M_F$ such that for all $x, y \in \mathbf{R}$,

$$|F(x) - F(y)| \le M|x - y| \tag{1.10}$$

Any implementation of F actually computes $v(F(v(x)))$, since the computer converts any real-number input x into its representable value $v(x)$, and at best produces a representable output for the exact value $F(v(x))$, even if intermediate steps are exact. To estimate this error we first break it into two parts:

$$\begin{aligned} |F(x) - v(F(v(x)))| &= |F(x) - F(v(x)) + F(v(x)) - v(F(v(x)))| \\ &\le |F(x) - F(v(x))| + |F(v(x)) - v(F(v(x)))|. \end{aligned}$$

For the first term, use the Lipschitz condition to write

$$|F(x) - F(v(x))| \le M|x - v(x)| \le M|x|\epsilon_f.$$

The second term may be estimated by

$$\begin{aligned} |F(v(x)) - v(F(v(x)))| &\le |F(v(x))|\epsilon_f \\ &= |F(x) + F(v(x)) - F(x)|\epsilon_f \end{aligned}$$

$$\begin{aligned}
&\le (|F(x)| + |F(v(x)) - F(x)|)\,\epsilon_f \\
&\le (|F(x)| + M|v(x) - x|)\,\epsilon_f \\
&\le (|F(x)| + M|x|\epsilon_f)\,\epsilon_f \\
&\le \left(1 + \frac{M|x|}{|F(x)|}\epsilon_f\right)|F(x)|\epsilon_f
\end{aligned}$$

These estimates combine to give $|F(x) - v(F(v(x)))| \le (M|x| + (1 + M\epsilon_f)|F(x)|)\,\epsilon_f$. Thus dividing by $F(x) \neq 0$ yields:

$$\left|\frac{F(x) - v(F(v(x)))}{F(x)}\right| \le \left(1 + M\epsilon_f + \frac{M|x|}{|F(x)|}\right)\epsilon_f. \tag{1.11}$$

Computing F is well-conditioned if $M|x| \approx |F(x)|$ and $M\epsilon_f \approx 1$, but can be ill-conditioned if $M|x| \gg |F(x)|$ or $M\epsilon_f \gg 1$.

If $x = (x_1, \ldots, x_n)$, and $F = F(x)$ is a function of several variables satisfying the Lipschitz condition

$$|F(x) - F(y)| \le \sum_k M_k|x_k - y_k|,$$

then by a similar argument we get

$$\left|\frac{F(x) - v(F(v(x)))}{F(x)}\right| < \left(1 + \frac{\sum_k M_k|x_k|}{|F(x)|}\right)\epsilon_f. \tag{1.12}$$

Such a multivariable inequality applies to products and quotients. For example, let $n = 2$ and put $F(x) = x_1x_2$. Then $M_1 \approx |x_2|$ and $M_2 \approx |x_1|$ for z within $\epsilon_f \ll 1$ of x. The right-hand side of Inequality 1.12 simplifies to

$$\left(1 + \frac{|x_2x_1| + |x_1x_2|}{|x_2x_1|}\right)\epsilon_f = 3\epsilon_f,$$

as in Inequality 1.8. The case of quotients is left as an exercise.

1.3 Exercises

1. Suppose a divides b and b divides a. Must $a = b$?
2. Write a computer program that finds the greatest common divisor of two integers a and b, assuming $a \ge b \ge 0$.
3. Prove that distinct primes are relatively prime.
4. Find the greatest common divisor of the three numbers 299 792 458, 6 447 287, and 256 964 964.
5. Find the quasi-inverse of 2301 modulo 19 687. (Hint: implement the extended Euclid algorithm first.)

6. Implement the Miller–Rabin primality test for odd N under the assumption that $2 < N < 341\,550\,071\,728\,321$.

7. Prove that integer overflow or underflow occurs in w-bit twos complement integer arithmetic if and only if the carry into the sign bit is different from the carry out of the sign bit.

8. Express the integer 14 600 926 (base 10) in hexadecimal.

9. Prove that if $p \in \mathbf{Z}$ is a prime number, then $\sqrt{p}$ is not a rational number.

10. Write a computer program to read an integer in decimal notation and then print its binary digits and its hexadecimal digits. (Hint: most computers expect decimal number inputs and thus have built-in functions to read them.)

11. Convert the approximation $\pi \approx 3.1415926535897932$ (base 10) into the nearest 8-digit hexadecimal fraction.

12. Using 52 bits to represent the mantissa in IEEE binary floating-point format, how many decimal digits of accuracy are obtained?

13. What will $\sum_{k=1}^{10^8} 1.0$ equal on the example computer on p. 23, which uses IEEE 32-bit floating-point arithmetic?

14. Write a program to read 32-bit IEEE binary floating-point format and print the number in scientific notation. Have it treat NaN, $\pm\infty$, and ± 0 properly and have it signal when the number is subnormal.

15. Derive Inequality 1.9 from Inequality 1.12.

16. Determine and prove whether the following computations are well-conditioned or ill-conditioned:

    a. $(x, y) \mapsto \sqrt{x^2 + y^2}$, for $x \neq 0$ and $y \neq 0$

    b. $x \mapsto x \log x$, for $x > 0$

    c. $x \mapsto \lfloor x \rfloor$

1.4 Further Reading

- ANSI/IEEE. *Standard for Binary Floating-Point Arithmetic.* Document 754-1985, catalog number SH 10116-NYF. ISBN 1-55937-653-8.

- Donald Knuth. *Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, Massachusetts, second edition, 1973. ISBN 0-201-03809-9.

- Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs.* Oxford University Press, New York, 2000. ISBN 0-19-512583-5.

- Herbert Schildt. *The Annotated ANSI C Standard: ANSI/ISO 9899-1990.* Osborne Mcgraw Hill, Berkeley, California, 1993. ISBN 0-07-881952-0.
- Douglas R. Stinson. *Cryptography: Theory and Practice.* CRC Press, Boca Raton, Florida, 1995. ISBN 0-8493-8521-0.