Mathematics and Algorithms for Computer Algebra

July 9, 2003

3: Integer and rational arithmetic

I now want to apply the basic notions of computational representations and abstract algebra that I have developed so far to concrete algorithms, and briefly to consider their complexity. As I explained earlier, underlying all of computer algebra are the integers, so I will begin by considering algorithms for performing arithmetic on integers *of unlimited magnitude*, which I will call *long* integers. It is then a very small step to extend this to arbitrary rational numbers.

I will not consider any approximate representations, such as rounded representations of the reals, because they are not fundamental to CA and they lie more within the domain of numerical analysis. Given algorithms to manipulate long integers it is a small extension to manipulate rounded real representations of unlimited magnitude and precision. I will not explicitly consider complex numbers because they are a trivial extension of the real numbers, and similarly arithmetic on the integers mod m, which as we have seen is very important, is essentially a trivial extension of integer arithmetic.

The definitive presentation of the material discussed in this set of notes is in Chapter 4 "Arithmetic" in Vol. 2 "Seminumerical Algorithms" of the classic book *The Art of Computer Programming* by Donald E. Knuth (Addison-Wesley, 1981), which henceforth I will refer to simply as "Knuth", and most of it is in section 4.3.1 "Multiple-precision arithmetic: The classical algorithms".

1 Representation of long integers

We have already seen how to do this in practice, by choosing a base B that conveniently fits into a word of memory, and then linking a sufficient

number of words together, either as a linked list or an array. There is a question about exactly what the base should be, to which we will return later. Any integer can be represented by its magnitude together with its sign. I will assume that the sign, which requires only one bit of information, is stored somewhere and can be accessed as required, and I will focus on the representation of non-negative integers. The manipulation of long integers essentially just formalizes the conventional techniques for performing integer arithmetic by hand using "long multiplication", "long division", etc. In order to understand the algorithms it is often useful to imagine the base B to be 10, even though the real intention is that the base will be more like 10^5 or 10^{10}.

The mathematics of this representation requires the lexicographic ordering once again, but now in a purely numerical context. Each digit (component) of a long (unlimited magnitude) integer is itself a (small) integer, so a (long) integer is represented by a set of digits (small integers), but this set has an order corresponding to the positional representation used when writing integers conventionally. Hence we need to formalize the ordering on such sets of digits.

1.1 Lexicographic order

Let $E_1, \ldots, E_r$ be ordered sets (e.g. each $\{0, 1, 2, \ldots, B-1\}$), and define the relation $<$ on the Cartesian product set $E = E_1 \times \cdots \times E_r$ of r-tuples of the form $x = (x_1, \ldots, x_r) \in E$ by

$$x < y \text{ if } x_i < y_i \text{ for the first index } i \text{ such that } x_i \neq y_i.$$

This is called the *lexicographic order* on E.

For example, taking $B = 10$ as in conventional decimal representation, and $r = 5$ to allow integers with up to 5 digits, this ordering would give

$$(1, 2, 3, ?, ?) < (1, 2, 4, ?, ?),$$

where ? represents *any* digit, or expressed more conventionally and in words: 123 hundred and anything is less than 124 hundred and anything.

[Note that if each E_i is an ordered set of variable identifiers together with 1 ordered so that $x_i > 1$ $\forall i$ and $(x_1, x_2, \ldots, x_r)$ denotes the *product* $(x_1 \times x_2 \times \cdots \times x_r)$ then E is the set of all monomials of total degree at most r, and this lexicographic ordering is exactly the same as that which I described less formally in the first set of notes.]

The set E together with the lexicographic order relation is called the *lexicographic product* of the sets E_i. If each E_i is totally ordered then so is E.

1.2 Existence and uniqueness of the representation

Theorem 1 *Let B be an integer > 1, and for k a strictly positive integer let $\mathbb{B}^k$ be the lexicographic product of k copies of the set $\mathbb{B} = \{0, 1, \ldots, B-1\}$. Then the map $f_k : \mathbb{B}^k \to \{0, 1, \ldots, B^k - 1\}$ defined by*

$$f_k : (d_0, d_1, \ldots, d_{k-1}) \mapsto \sum_{i=0}^{k-1} d_i B^{k-i-1}$$

is an isomorphism.

Proof is by induction on k. □

This shows how to go from a representation to the integer that it represents. To go the other way we need the observation that for any integer $a > 0$ there is a smallest integer $k > 0$ such that $a < B^k$. Then according to the above theorem a has a unique representation of the form

$$a = (d_0, d_1, \ldots, d_{k-1})_B, \quad (d_0 \neq 0).$$

The (small) integers d_i are called the *digits* of a in base B; the base is normally only indicated (as a subscript) when necessary to avoid ambiguity.

Finally, the following bounds will be useful later.

Proposition 2 *If the positive integer a admits the following representation in base B:*

$$a = \sum_{i=0}^{k} a_i B^i, \quad \textit{with } a_k \neq 0,$$

then

$$B^k \leq a_k B^k \leq a \leq B^{k+1} - 1.$$

[Beware of the change of notation here!]

Proof The only non-trivial inequality is the last one, which follows from the observation that

$$a_i \leq B - 1 \;\Rightarrow\; a \leq (B-1)(1 + B + \cdots + B^k) = B^{k+1} - 1.$$

□

1.3 Number bases

The most commonly used bases are 10, giving the *decimal* or *denary* system, 2 giving the *binary* system, 8 giving the *octal* system and 16 giving the *hexadecimal* system. Modern computers normally use binary internally, although because binary representation generates rather long strings of digits it is common for computer programmers to use hexadecimal or octal notation. The smallest addressable amount of memory is usually 8 bits, called a *byte*, and one hexadecimal digits corresponds to 4 bits, which is sometimes called a *nibble* (or *nybble*) (because "nibble" means a small "bite" in the more conventional context of eating food). Therefore one byte is conveniently represented by two hexadecimal digits. However, we only have conventional symbols for 10 digits, whereas hexadecimal representation needs 16 digits, so these are represented as $\{0, 1, 2, \ldots, 9, a, b, \ldots, f\}$. Octal representation has the advantage that it requires only (the first 8) conventional digit symbols.

(In fact, many computers can also work in decimal internally using a representation called *binary coded decimal* (BCD), in which nybbles are used to store only decimal digits. If not much arithmetic is performed then BCD avoids the input-output overhead of converting from decimal to binary and back again, and may therefore be more efficient, but there is no floating-point analogue of BCD and it is used in commerce rather than in science.)

When a representation requires more memory than is available it is said to *overflow*, which can easily happen in the conventional representation of numbers using a fixed number of words. In our flexible multi-word representation it would correspond to completely running out of memory, which is certainly possible. However, it is important to avoid conventional overflow during arithmetic calculations with multi-word integers. Addition (and hence subtraction) of two integers in any base can produce at most one extra digit, and so a choice of base that leaves at least one (small) digit position free in each word of the representation is necessary. Thus if the word size of a computer is n bits the base should use at most $n - 1$ of them, and so we require $B \leq 2^{n-1}$. This implies a maximum digit size of $2^{n-1} - 1$, hence we can add two maximal digits to get $2^n - 2$ which is less than the maximum (small) integer that can be stored in an n-bit word, namely $2^n - 1$.

This choice of base is the most storage-efficient among reasonable choices, but it causes difficulty in multiplication, because the product of two $(n-1)$-bit digits can have up to $2n-2$ bits! Using only half of the available bits to avoid this problem does not make efficient use of memory, so the solution is

to take care when designing the multiplication algorithm, to which we will return below.

1.4 Converting between representation and long integer

To convert from a representation $a = (a_0, a_1, \ldots, a_{k-1})$ to the integer that it represents is simply a matter of evaluating the sum

$$a = \sum_{i=0}^{k-1} a_i B^{k-i-1}.$$

Note that this is the same as the standard numerical task of evaluating a (univariate) polynomial, and the efficient way to do it is exactly the same. For example, to numerically evaluate the polynomial

$$a_0 x^3 + a_1 x^2 + a_2 x + a_3$$

one writes it in Horner's nested form as

$$((a_0 x + a_1)x + a_2)x + a_3,$$

which (generally) involves the minimum number of multiplications, and (as always) evaluate from the inside of the nest of parentheses outwards. In the general case, this evaluation scheme is expressed by the following *algorithm*. I will write algorithms in an algorithmic pseudo-language that is similar to Pascal and REDUCE, and not unlike Maple, C, etc.

input: B integer > 1, k, a_0 integers > 0, $a_1, \ldots, a_{k-1}$ integers ≥ 0
$a := a_0$;
for $i := 1$ to $k - 1$ do $a := aB + a_i$.
output: the long integer a

This algorithm is interesting in theory, and it is useful (and used) in practice when working with small integers, but for long integers we cannot actually use it, because the long integer has no computational significance other than as its representation – in other words, for us a long integer *is* its representation, and what we need is algorithms to manipulate long integers by manipulating their representations. Nevertheless, such algorithms are based on this first and simplest algorithm. For example, there is a need for algorithms to convert integer representations between different bases, and in particular to convert between base 10 and the internal base for input and output purposes.

The conversion of a long integer into its digit representation has a similar status to the above conversion the other way. It is perhaps most obvious from the explicit Horner form above that

$$a = Bq + a_{k-1}, \quad \text{where } 0 \leq a_{k-1} < B,$$

so that under integer division the last digit a_{k-1} is the remainder when the number a is divided by the base B, i.e.

$$a_{k-1} = a \bmod B.$$

Then the quotient q has the form

$$q = \sum_{i=0}^{k-2} a_i B^{k-i-2},$$

of which the last digit again is given by

$$a_{k-2} = q \bmod B.$$

This suggests a conversion procedure, but writing it as an explicit algorithm is not quite as easy as the opposite conversion, because we do not know in advance how many digits will be generated. Therefore we cannot simply loop through all the digits, and instead we must use a more general condition on the number being decomposed into digits to stop the loop, thus:

input: a integer ≥ 0, B integer > 1
$i := 0$;
while $a > 0$ do
begin
    $x_i := a$ mod B;
    $i := i + 1$;
    $a := a$ div B
end.
output: the representation $(x_{k-1}, \ldots, x_1, x_0)_B$

Note that this algorithm destroys the value of the variable a; in practice one would probably make it local so that this would not matter – otherwise a copy should be used instead. I have used div to denote the quotient in an

integer division operation, as provided in Pascal, symmetricaly with mod for the remainder.[1]

Note also that this algorithm naturally generates the digits in the "wrong order", i.e. from the right in increasing weight order, so that $x_i = a_{k-i-1}$. In order to use essentially the same iterative algorithm to generate the digits in the "right order", i.e. from the left in decreasing weight order as would be necessary for example to print out the number, it is necessary to first compute the number of digits k. This is $\lfloor \log_B a \rfloor + 1$,[2] but it is better to compute it simply by counting in a preliminary loop, thus:

input: a integer ≥ 0, B integer > 1
$k := 1$; $x := 1$;
while $a > xB$ do
begin
    $k := k + 1$;
    $x := xB$
end;
for $i := 0$ to $k - 1$ do
begin
    $a_i := a$ div x;
    $a := a$ mod x;
    $x := x$ div B
end.
output: the representation $(a_0, a_1, \ldots, a_{k-1})_B$

1.5 Iteration versus recursion

The above iterative algorithm to compute digits in decreasing weight order is inelegant, whereas a recursive algorithm could naturally generate them in increasing weight order. In fact, it is frequently the case that a recursive algorithm follows the opposite order to an iterative algorithm. This is essentially because recursion naturally involves two passes in opposite orders through the data: an "inward" pass that stacks recursive invocations, followed by an "outward" pass that unstacks them.

[1] Unfortunately, the normal (algebraic) user mode of REDUCE does not provide either of these keywords, although both facilities exist internally and could easily (and I believe should) be made more readily available!

[2] The *floor* function $\lfloor \cdot \rfloor : \mathbb{R} \to \mathbb{Z}$ is defined such that $\lfloor x \rfloor = n$ is the largest integer not greater than x, i.e. $n \leq x < n + 1$.

Using the notation $\mathrm{rep}(a;B) = (a_0, a_1, \ldots, a_{k-1})_B$ to denote the k-tuple representation of the integer a in base B, the observation that

$$a = \sum_{i=0}^{k-1} a_i B^{k-i-1} = Bq + a_{k-1}$$

where

$$q = \sum_{i=0}^{k-2} a_i B^{k-i-2} = a \text{ div } B, \quad a_{k-1} = a \bmod B,$$

gives the recursive formulation

$$\mathrm{rep}(a;B) = \mathrm{rep}(a \text{ div } B;B) \| (a \bmod B) \text{ if } a \neq 0,$$

$$\mathrm{rep}(0;B) = (0),$$

where the "concatenation operator" $\|$ is defined by

$$(x_1, x_2, \ldots, x_{r-1}) \| (x_r) = (x_1, x_2, \ldots, x_{r-1}, x_r),$$

$$(0) \| (x) = (x).$$

Since the above recursive formulation was in terms of building a list (i.e. a k-tuple), here is a more algorithmic example of a recursive procedure (one that calls itself) to *output* the digits in the correct order. Notice that it is very succinct (short) and elegant, and that it *must* have a name (so that it can call itself):

```
procedure rep(a, B);
if a ≠ 0 then
begin
    rep(a div B, B);
    output a mod B
end.
```

1.6 Comparing two long integers

If the two integers have opposite signs then the comparison operation is trivial and requires only the sign information. I will consider only the case that they are both positive; the case that they are both negative trivially requires the comparison to be reversed. Let $a = (a_m, \ldots, a_0)_B$ and $b = (b_n, \ldots, b_0)_B$ be representations in base B of two positive (long) integers, such that $a_m, b_n \neq 0$.

It is convenient to express the comparison in terms of the function

$$\text{sign}(x) = \begin{cases} +1 & \text{if } n > 0, \\ 0 & \text{if } n = 0, \\ -1 & \text{if } n < 0. \end{cases}$$

We assume that this function and arithmetic are supplied for small integers, and that B, m, n are small integers.

The comparison algorithm is a direct consequence of the representation theorem (Theorem 1) and lexicographic ordering:

input: $a = (a_m, \ldots, a_0)_B$, $b = (b_m, \ldots, b_0)_B$, $a_m, b_n \neq 0$
if $m \neq n$ then $r := \text{sign}(m - n)$ else
begin
    $i := m$;
    while $a_i = b_i$ do $i := i - 1$;
    if $i = -1$ then $r := 0$ else $r := \text{sign}(a_i - b_i)$
end.
output: $r = \text{sign}(a - b)$

2 Addition and subtraction of long integers

I will consider only the addition and subtraction of *pairs* of integers; the case of many numbers can trivially be expressed as repeated operations on pairs.

2.1 Two positive integers

Let $a = (a_{m-1}, \ldots, a_0)_B$ and $b = (b_{n-1}, \ldots, b_0)_B$ be representations in base B of two positive (long) integers with respectively m and n digits; then

$$a = \sum_{i=0}^{m-1} a_i B^i \quad \text{and} \quad b = \sum_{i=0}^{n-1} b_i B^i.$$

Note that for addition and multiplication it is convenient to sum in ascending weight order, which is the reverse of the representation order. If the representation is implemented as arrays then this presents no difficulty at all; if the representation uses lists then either they must first be reversed, or a recursive summation algorithm could be used that is slightly more subtle than the iterative version presented below.

If $m \geq n$ then the sum is formally

$$c = a + b = \sum_{i=0}^{m-1} (a_i + b_i)B^i \quad \text{where } b_n = b_{n+1} = \cdots = b_{m-1} = 0.$$

However it is generally not the case that the representation of c in base B is simply $c = (a_{m-1} + b_{m-1}, \ldots, a_0 + b_0)_B$ because it may happen that $a_i + b_i \geq B$, in which case a *carry* to the digit position having next higher weight is generated. The carry can be only 0 or 1 (in any base) because by definition

$$a_i, b_i \leq B - 1 \;\Rightarrow\; a_i + b_i \leq 2B - 2 = 1B + (B - 2).$$

In the following algorithm, r denotes the carry at each stage, and it is assumed that a small amount of pre-processing has been done to ensure that b has no fewer digits than a by exchanging a and b if necessary.

input: $a = (a_{m-1}, \ldots, a_0)_B$, $b = (b_{n-1}, \ldots, b_0)_B$, $a, b \geq 0$, $m \geq n$
$r := 0$; {carry forward}
$k := m - 1$; {length of sum}
for $i := 0$ to $m - 1$ do
begin
    $s := a_i + b_i + r$;
    if $s < B$ then begin $c_i := s$; $r := 0$ end
        else begin $c_i := s - B$; $r := 1$ end
end;
if $r > 0$ then begin $k := m$; $c_m := 1$ end.
output: $c = a + b = (c_k, \ldots, c_0)_B$

2.2 Two integers of arbitrary sign; subtraction

If a and b have the same sign then

$$c = a + b = \text{sign}(a)\,(|a| + |b|)$$

and the previous addition algorithm can be used. If a and b have opposite signs then two (and only two) cases need to be distinguished:

- if $|a| \geq |b|$ then $c = a + b = \text{sign}(a)\,(|a| - |b|)$;
- if $|a| < |b|$ then $c = a + b = \text{sign}(b)\,(|b| - |a|)$.

Hence it is necessary to be able to compare $|a|$ and $|b|$, for which we already have an algorithm (taking an absolute value is trivial – just reset the sign indicator to positive), and to be able to subtract one positive integer from another that is no larger. Assuming as before that the shorter integer b is *padded* with leading zeros, the algorithm can be almost identical to the addition algorithm, except that the carry is subtracted rather than added:

input: $a = (a_{m-1}, \ldots, a_0)_B$, $b = (b_{n-1}, \ldots, b_0)_B$, $a \geq b \geq 0$ ($\Rightarrow$ $m \geq n$)
$r := 0$; {carry forward}
for $i := 0$ to $m - 1$ do
begin
    $s := a_i - b_i - r$;
    if $s \geq 0$ then begin $c_i := s$; $r := 0$ end
        else begin $c_i := s + B$; $r := 1$ end
end.
output: $c = a - b = (c_{m-1}, \ldots, c_0)_B$

Note that there can be no carry beyond the digits of a by the assumption that $a \geq b$, which makes this algorithm slightly simpler than the addition algorithm.

Subtraction itself is now trivially implemented in terms of addition as

$$a - b = a + (-b);$$

note that it is necessary to use the full addition algorithm to accommodate arbitrary signs for a and b.

2.3 Complexity of addition and subtraction

As I discussed in the first set of notes, the natural choice for the complexity of an algorithm is the number of operations on small integers. In both the above algorithms there is only one loop which performs m iterations (assuming $m \geq n$). We can conveniently lose the details of what happens in the loop, and precisely which operations we count, in the "O" notation, which ignores constant factors. Therefore, a reasonable estimate of the complexity bound is $O(\max\{m, n\})$.

As a worst-case bound this cannot be replaced by $O(\min\{m, n\})$, by modifying the algorithm to stop the loop when the digits of b are exhausted, because it is possible for a carry to propagate right through the remaining

digits of a, as illustrated by the following worst possible example of this phenomenon for addition:

$$a = \sum_{i=0}^{m-1} (B-1)B^i, \quad b = 1, \quad c = a + b = B^m,$$

because

$$a = (B-1)(B^{m-1} + B^{m-2} + \cdots + B + 1) = B^m - 1.$$

A similar worst case can occur in a non-trivial subtraction.

3 Multiplication of long integers

I will consider only pairs of positive integers because it is trivial to accommodate arbitrary signs in a product, and to generalize to a product of many integers. With a and b as above, $m \geq n$, their product is given by

$$c = ab = \left(\sum_{i=0}^{m-1} a_i B^i\right)\left(\sum_{j=0}^{n-1} b_j B^j\right) = \sum_{h=0}^{m+n-2} \left(\sum_{i+j=h} a_i b_j\right) B^h.$$

The limits on the inner sum can be arranged so that non-existent digits are not accessed, by combining the relations

$$0 \leq i \leq m-1,$$

and

$$0 \leq j \leq n-1, \quad i = h - j \;\Rightarrow\; h - n + 1 \leq i \leq h.$$

However, the inner sum can attain the maximum value of $\alpha(B-1)^2$, where α depends on m, n in a fairly complicated way but is clearly bounded by $m + n - 1$. Hence, when converting the outer sum into a representation in base B, the carry can be arbitrarily large, and so must be handled in a more general way than in the addition and subtraction algorithms. Nevertheless,

$$a < B^m, \quad b < B^n \;\Rightarrow\; c = ab < B^{m+n},$$

and hence the number of digits in the product cannot be more than $m + n$.

With this analysis, the double sum formulation translates directly into the following algorithm:

input: $a = (a_{m-1}, \ldots, a_0)_B$, $b = (b_{n-1}, \ldots, b_0)_B$, $a, b \geq 0$
$r := 0$; {carry forward}
$k := m + n - 1$; {length of product}
for $h := 0$ to $m + n - 2$ do
begin
    $s := r$;
    for $i :=$ max$(0, h - n + 1)$ to min$(h, m - 1)$ do $s := s + a_i b_{h-i}$;
    $c_h := s$ mod B;
    $r := s$ div B
end;
if $r \neq 0$ then begin $k := k + 1$; $c_{k-1} := r$ end.
output: $c = ab = (c_{k-1}, \ldots, c_0)_B$

This algorithm is a direct translation of the nested sum formula, which is itself a reformulation of the product of two sums, hence all three involve the same number of multiplications, which is most easily seen from the product of sums to be mn. The number of other operations is clearly also bounded by this number, and therefore the complexity is $O(mn)$.

When directly multiplying two digits $a_i, b_j \leq B - 1$ there is an unavoidable possibility of the product being $(B-1)^2$, which whilst being $\geq B$ (for $B \geq 3$) is also $< B^2$, so that whilst it may not fit into one small integer, it can be represented using two, which I will call a *double* small integer. However, the above algorithm also *sums* such products, and thereby can produce intermediate values that do not fit into a double small integer, and moreover are not bounded independently of m, n. This can be avoided by re-writing the double sum as

$$c = ab = \sum_{j=0}^{n-1} \left(\sum_{i=0}^{m-1} a_i b_j B^{i+j} \right),$$

which puts the powers of B *inside* the inner sum and thereby spreads out the products $a_i b_j$ so that within the inner sum they are not added but only stored. This formulation translates into the following algorithm:

input: $a = (a_{m-1}, \ldots, a_0)_B$, $b = (b_{n-1}, \ldots, b_0)_B$, $a, b \geq 0$
for $i := 0$ to $m - 1$ do $c_i := 0$; {if necessary}
$k := m + n$; {length of product}
for $j := 0$ to $n - 1$ do
begin
    $r := 0$; {carry forward}

```
    for i := 0 to m − 1 do
    begin
        s := a_i b_j + c_{i+j} + r;
        c_{i+j} := s mod B;
        r := s div B
    end;
    c_{j+m} := r; {hence no need to initialize}
end.
if r = 0 then k := k − 1;
```

output: $c = ab = (c_{k-1}, \ldots, c_0)_B$

Note that if the variables storing the digits of the product c cannot be assumed to be zero at the start of the algorithm (which depends on details of an implementation) then it is necessary to explicitly clear only the first m, because the carry from each evaluation of the inner sum assigns directly to the next higher digit position in c.

To prove that this algorithm does not overflow a double small integer representation for the intermediate sum s or a small integer representation for r, note that

$$s = a_i b_j + c_{i+j} + r \leq (B-1)^2 + (B-1) + r, \quad r = s \text{ div } B.$$

Assuming $s < B^2$ then $r < B$, and hence consistently

$$s \leq (B-1)^2 + (B-1) + (B-1) < (B-1)^2 + 2(B-1) + 1 = B^2.$$

We still have the problem that s generally requires a double small integer representation, whereas the motivation for representing a long integer as a sequence of small integers was that the small integer arithmetic could be performed directly by the computer hardware. In fact, many processors support double-word arithmetic, and therefore there is no problem, unless the small integer representation itself was based on a double word. If necessary, double-small-integer arithmetic could be implemented in software as a simple special case of the implementation of long integer arithmetic, for example by choosing B to be a perfect square and representing a small integer $u < B$ in the form $u = u_0\sqrt{B} + u_1$, $u_0, u_1 < \sqrt{B}$, so that $u_i u_j < B$ and this arithmetic really can be performed using only small integers. This software implementation would increase the complexity of the algorithm, but only by a constant factor, so that it would still be $O(mn)$.

4 Euclidean division

Division of numbers is much harder than any of the other arithmetic operations, in much the same way that integration is much harder than differentiation. In the same way that multiplication by an integer corresponds to repeated addition of ring elements, so division corresponds to repeated subtraction, and it could be implemented that way, but for large integers one would prefer a more efficient approach! Many books on computer algebra are rather vague about integer division. I will present a complete but simplified algorithm; for the most efficient version see Knuth.

Let a and b be two long integers having respectively m and n digits in base B representation. Because division proceeds from the left it is convenient to number the digits from the left as we did originally, to give

$$a = (a_0, a_1, \ldots, a_{m-1})_B, \quad b = (b_0, b_1, \ldots, b_{n-1})_B.$$

Then the problem is to divide a by b to obtain a quotient q and remainder r satisfying the integer division relation (which is a special case of the Euclidean division relation):

$$a = bq + r, \quad 0 \leq r < |b|.$$

One need not necessarily insist that the remainder r be positive in all cases, provided q and r are uniquely defined, as I discussed in Notes 2. However, I will consider only strictly positive integers a and b – the generalization to include all integers is a technical detail.

If $a < b$ then trivially $q = 0$ and $r = a$, so assume that $a > b$ (and hence $m \geq n$). Because $B^{m-1} \leq a < B^m$ and $B^{n-1} \leq b < B^n$ it follows that

$$q \leq a/b < B^m/B^{n-1} = B^{m-n+1}$$

and hence the length k of q satisfies $k \leq m - n + 1$.

Long division of a by b determines the digits of the quotient q one by one from the left (i.e. in decreasing weight order). The leading significant digit q_0 is determined by dividing b into the number consisting of the smallest leading subset of the digits of a for which the division is possible (i.e. gives a non-zero quotient). At most the leading $n + 1$ digits of a are required: if $a_0 > b_0$ then the leading n digits suffice, if $a_0 = b_0$ then the leading n *may* suffice; otherwise the leading $n + 1$ digits are required. However, it is easiest to add a leading 0 to the front of a and allow the leading digit of the quotient to be 0, in which case we can *always* start with the leading $n + 1$

digits of a. The leading quotient digit q_0 times b is then subtracted from the leading $n+1$ digits of a, to leave a remainder r that is guaranteed to be less than b. Digit $n+1$ of a is now appended to r to make it into an $n+1$-digit number, which when divided by b is guaranteed to have a quotient q_1 such that $0 \le q_1 < B$. The process is repeated until all digits of a have been processed, when the final value of r is the overall remainder.

Then the difficult part of the algorithm is to determine the quotient Q of an $n+1$-digit integer by an n-digit integer, which is quaranteed to satisfy $0 \le Q < B$. Let us assume that we have a suitable sub-algorithm, which we will develop later, to perform this task; then the overall division algorithm looks like this:

input: $a = (a_0, \ldots, a_{m-1})_B$, $b = (b_0, \ldots, b_{n-1})_B$, $a \ge b \ge 0$ ($\Rightarrow$ $m \ge n$)
$k := m - n + 1$; {length of quotient}
{Initial $n+1$-digit remainder:}
$r := (0, a_0, \ldots, a_{n-1})_B$;
for $i := 0$ to $k - 1$ do
begin
    {Determine next digit of quotient:}
    $q_i := \lfloor r/b \rfloor$;
    $r := r - q_i b$; {at most n significant digits}
    {Append next digit of a:}
    if $i \ne k - 1$ then $r := (r_0, \ldots, r_{n-1}, a_{(m-k+1)+i})_B$;
end;
output: $q = (q_0, \ldots, q_{k-1})_B$, $r = (r_0, \ldots, r_{n-1})_B$ $(a = bq + r,\ 0 \le r < b)$

The leading digit q_0 of the quotient q may be zero (in which case it can be discarded), and the length of the remainder r can be anything between 0 and the length n of b.

Before proceeding further, we can now bound the complexity of integer division. The loop determines one digit of q on each iteration, and the length k of q satisfies $k \le m - n + 1$, so that the loop is executed at most $m-n+1$ times. Because $0 \le q_i \le B-1$, q_i could in principle be determined by testing all $B - 1$ possible values in a time that is independent of m, n (although we will see below that we can do a lot better than that). The time to multiply the integer b of length n by the digit q_i is clearly $O(n)$, and the time to subtract the result from r is clearly also $O(n)$. Therefore, the complexity of the whole algorithm is $O((m-n)n)$, which is the same as the time required to compute the product bq. Note that the details of the algorithm, such as precisely how each digit in the quotient is determined,

do not affect the order of the asymptotic time complexity; however, they do affect the constant factor that is suppressed by the "O" notation but can be significant in practice, because long integers of practical importance typically have no more than a few hundred decimal digits, which in a base of say 10^{10} corresponds to $m, n \approx 10$, which is not very asymptotic!

4.1 Determining digits of the quotient

With a minor shift of notation, the problem is the following:

> Let $u = (u_0, u_1, \ldots, u_n)_B$ and $v = (v_1, v_2, \ldots, v_n)_B$, $v_1 \neq 0$, be non-negative integers in base-B representation, such that $u/v < B$. Find an algorithm to determine $Q = \lfloor u/v \rfloor$.

The condition that $u/v < B$ is equivalent to

$$u/B < v \;\Rightarrow\; \lfloor u/B \rfloor < v \;\Rightarrow\; (u_0, u_1, \ldots, u_{n-1})_B < (v_1, v_2, \ldots, v_n)_B,$$

i.e. the leading n digits of u do not have a non-zero quotient with v. We ensured this initially by prepending a leading 0 to the dividend a; it is ensured subsequently by the integer division property which ensures that the remainder r is strictly less than the divisor b. Furthermore, Q is the unique quotient such that

$$u = vQ + R, \quad 0 \leq R < v.$$

It will not always be the case that $u_0 \geq v_0$, but it will always be the case that $(u_0, u_1)_B \geq v_0$, so their quotient is a plausible first guess at Q. Therefore, since we know that $Q < B$, let us guess

$$\hat{Q} = \min\left(\left\lfloor \frac{u_0 B + u_1}{v_1} \right\rfloor, B - 1\right).$$

It is not obvious, but we will now prove that $\hat{Q}$ is guaranteed to be very close to Q provided v_1 is large enough (which can easily be arranged by rescaling both u and v). The proofs use the properties of the floor function that $\lfloor x \rfloor > x - 1$ and $\lfloor x \rfloor \leq x$.

Surprisingly, $\hat{Q}$ can never be too small:

Proposition 3 *With the definitions above, $\hat{Q} \geq Q$.*

Proof It is obviously true if $\hat{Q} = B - 1$. Otherwise

$$\hat{Q}v_1 > u_0B + u_1 - v_1 \;\Rightarrow\; \hat{Q}v_1 \geq u_0B + u_1 - v_1 + 1.$$

It then follows that

$$\begin{aligned} u - \hat{Q}v &\leq u - \hat{Q}v_1B^{n-1} \\ &\leq (u_0B^n + \cdots + u_n) - (u_0B + u_1 - v_1 + 1)B^{n-1} \\ &= u_2B^{n-2} + \cdots + u_n - B^{n-1} + v_1B^{n-1} \\ &< v_1B^{n-1} \leq v, \end{aligned}$$

i.e. $u - \hat{Q}v < v$. But Q is the smallest integer such that $u - Qv = R < v$ because also $R \geq 0$, hence $\hat{Q} \geq Q$. □

We now prove that also $\hat{Q}$ cannot be much too big, by contradiction. Suppose that $\hat{Q} \geq Q + 3$. By definition

$$\hat{Q} \leq \frac{u_0B + u_1}{v_1} = \frac{u_0B^n + u_1B^{n-1}}{v_1B^{n-1}} \leq \frac{u}{v_1B^{n-1}} < \frac{u}{v - B^{n-1}}$$

because $v - v_1B^{n-1} < B^{n-1}$. The case $v = B^{n-1} \;\Rightarrow\; v = (1, 0, 0, \ldots, 0)_B \;\Rightarrow$ $Q = \lfloor u/v \rfloor = u_0B + u_1 = \hat{Q}$, which is ruled out by the assumption $\hat{Q} \geq Q + 3$. Moreover, this assumption and the relation $Q > (u/v) - 1$ imply that

$$3 \leq \hat{Q} - Q < \frac{u}{v - B^{n-1}} - \frac{u}{v} + 1 = \frac{u}{v}\left(\frac{B^{n-1}}{v - B^{n-1}}\right) + 1.$$

Therefore,

$$\frac{u}{v} > 2\left(\frac{v - B^{n-1}}{B^{n-1}}\right) \geq 2(v_1 - 1).$$

Finally, $\hat{Q} \leq B - 1$ by definition, so that

$$B - 4 \geq \hat{Q} - 3 \geq Q = \lfloor u/v \rfloor \geq 2(v_1 - 1)$$

and hence $v_1 < \lfloor B/2 \rfloor$. This result together with the previous proposition prove:

Theorem 4 *If* $v_1 \geq \lfloor B/2 \rfloor$ *then* $\hat{Q} - 2 \leq Q \leq \hat{Q}$.

Hence the only possible error in $\hat{Q}$ is that it can over-estimate the true quotient Q by at most 2, and this is *independent* of the base B. Hence the $B-1$ possible values that Q might take have been reduced to just 3.

The requirement that $v_1 \geq \lfloor B/2 \rfloor$ is analogous to a normalization condition on the representation of v, which essentially states that the leading significant digit should be non-zero, although here we have a more stringent requirement. A simple way to meet the requirement is to multiply both u and v by $d = \lfloor B/(v_1+1) \rfloor$, which satisfies $\lfloor B/2 \rfloor \leq dv_1 \leq B-1$.

To use this estimate of the quotient, the overall division algorithm needs a pre-processing step in which both a and b are multiplied by $d = \lfloor B/(b_0+1) \rfloor$, and a post-processing step in which the remainder r *only* is divided by d. (Note that it must be the case that $d \,|\, r$.) Then an algorithm to compute $Q = \lfloor u/v \rfloor$ under the assumptions explained at the beginning of this subsection is the following:

input: $u = (u_0, u_1, \ldots, u_n)_B$, $v = (v_1, v_2, \ldots, v_n)_B$, $v_1 \neq 0$, $u, v > 0$
$Q := \hat{Q}$; {as above}
$R := u - Qv$;
while $R < 0$ do {at most 3 iterations}
begin
    $Q := Q - 1$;
    $R := R + v$
end.
output: Q, R such that $u = vQ + R,\ 0 \leq R < v$

This algorithm can be inserted into the main division algorithm presented earlier. Knuth gives a slightly more sophisticated and faster version.

5 Faster multiplication

The straightforward multiplication algorithm described earlier had complexity $O(mn)$ when multiplying two integers a and b with respectively m and n digits. It is interesting to know whether this complexity is optimal, and in fact it is not. A simple algorithm with lower asymptotic complexity is the following, essentially as described by Knuth.[3]

[3]The essence of the algorithm was apparently first suggested by A. Karatsuba in *Doklady Akad. Nauk SSSR* **145** (1962), 293–294 [English translation in *Soviet Physics–Doklady* **7** (1963), 595–596].

Suppose that we wish to multiply two long integers u, v each with $2N$ digits in base B. [In practice, this means choosing N to be the smallest integer so that $2N \geq m, n$, and padding a and b with leading zeros as necessary.] Then write u, v in the form

$$u = B^N u_1 + u_0, \quad v = B^N v_1 + v_0,$$

which is equivalent to representing them in base B^N; $u_1 = (u_{2N-1}, \ldots, u_N)_B$ is the more significant half of u, $u_0 = (u_{N-1}, \ldots, u_0)_B$ is the less significant half, and similarly for v. Then

$$\begin{aligned} uv &= B^{2N} u_1 v_1 + B^N (u_1 v_0 + v_1 u_0) + u_0 v_0 \\ &= (B^{2N} + B^N) u_1 v_1 + B^N (u_1 - u_0)(v_0 - v_1) + (B^N + 1) u_0 v_0. \end{aligned}$$

This formula replaces one multiplication of $2N$-digit numbers by three multiplications of N-digit numbers. This will probably be slightly faster, but the main use of the formula is to apply it recursively to perform the N-digit multiplications, etc. This implies that N should be chosen initially to be a power of 2.

If $T(N)$ denotes the time to perform an N-digit multiplication, then the above transformation implies that

$$T(2N) \leq 3T(N) + cN$$

for some constant c, because one $2N$-digit multiplication has been replaced by three N-digit multiplication and some additions and subtractions, each of which require time $O(N)$. Now if the technique is applied recursively k times we can apply the above complexity bound recursively k times to get

$$\begin{aligned} T(2^k) &= T(2.2^{k-1}) \\ &\leq 3T(2^{k-1}) + c2^{k-1} \\ &\leq 3^2 T(2^{k-2}) + c(3.2^{k-2} + 2^{k-1}) \\ &\vdots \\ &\leq 3^{k-1} T(2) + c(3^{k-2}.2 + \cdots + 3.2^{k-2} + 2^{k-1}). \end{aligned}$$

If c is chosen to be large enough that $T(2) \leq c$ then

$$\begin{aligned} T(2^k) &\leq c(3^{k-1} + 3^{k-2}.2 + \cdots + 3.2^{k-2} + 2^{k-1}) \\ &= c(3^{k-1} + 3^{k-2}.2 + \cdots + 3.2^{k-2} + 2^{k-1})(3-2) = c(3^k - 2^k) \end{aligned}$$

(which for $k = 1$ consistently gives $T(2) \leq c$). Therefore

$$T(n) \leq T(2^{\lceil \lg n \rceil}) \leq c(3^{\lceil \lg n \rceil} - 2^{\lceil \lg n \rceil}) < 3c \cdot 3^{\lg n} = 3cn^{\lg 3}$$

is an estimate of the complexity of multiplying two n-digit numbers.[4]

Hence, this technique of splitting a problem into two similar subproblems, and applying the splitting recursively (which is a standard technique for improving algorithms), has reduced the asymptotic complexity of multiplying two n-digit numbers from $O(n^2)$ to $O(n^{\lg 3}) \approx O(n^{1.585})$.

One can take this idea further, again following Knuth, and chop each number into $r + 1$ pieces rather than only 2, assuming that the numbers have $(r+1)N$ digits (in base B). Then formally replacing the new base B^N by a variable x associates each number

$$u = u_r B^{Nr} + \cdots + u_1 B^N + u_0$$

with the polynomial

$$u(x) = u_r x^r + \cdots + u_1 x + u_0,$$

and similarly for v. Define the polynomial

$$w(x) = u(x)v(x) = w_{2r} x^{2r} + \cdots + w_1 x + w_0.$$

Then because the numbers u, v can be recovered from their associated polynomials as $u = u(B^N)$, $v = v(B^N)$, we have that $uv = w(B^N)$. The coefficients of the degree-$2r$ polynomial $w(x)$ can be found as linear combinations of its values at $2r+1$ distinct values of x. (They are the solution of a system of $2r + 1$ linear equations, which can be solved formally before being evaluated numerically, or equivalently the Lagrange interpolation formula can be used.) The values of $w(x)$ are found as $u(x)v(x)$. The complexity of this operation is determined by the size of the coefficients, which have N digits (in base B); the interpolation is $O(N)$ and the multiplications require $T(N)$ operations. Then a single application of this technique gives

$$T((r + 1)N) \leq (2N + 1)T(N) + cN.$$

Applying this recursively as before leads to

$$T(n) \leq cn^{\log_{r+1}(2r+1)} < cn^{\log_{r+1}((r+1)2)} = cn^{1+\log_{r+1} 2}$$

[4] lg denotes the binary logarithm function $\log_2$; the *ceiling* function $\lceil \cdot \rceil : \mathbb{R} \to \mathbb{Z}$ is defined such that $\lceil x \rceil = n$ is the smallest integer n not less than x, i.e. $n - 1 < x \leq n$; $3^{\lg n} = n^{\lg 3}$ because $\lg 3 \lg n = \lg n \lg 3$.

as an estimate of the complexity of multiplying two n-digit numbers. Hence, we have proved the following:

Theorem 5 *Given $\varepsilon > 0$, there exists a multiplication algorithm such that the number of elementary operations $T(n)$ needed to multiply two n-digit integers satisfies*

$$T(n) < c(\varepsilon)n^{1+\varepsilon},$$

for some constant $c(\varepsilon)$ independent of n.

Unfortunately, this technique is not as good as it appears at first sight, because the inherent complication of the algorithm hidden in $c(\varepsilon)$ becomes very large as $\varepsilon \to 0$, i.e. $r \to \infty$, and in fact there are algorithms that make better use of the underlying interpolation-evaluation technique. The algorithms currently considered to be the best use finite Fourier transform techniques developed by Schönhage and Strassen, and for practical values of the size n of the integers give complexity essentially linearly proportional to n. However, these questions are of mainly theoretical interest, because the numbers that occur in CA are not normally large enough to make these sophisticated methods significantly better than the straightforward algorithm presented earlier.

6 Computation of powers

We have considered algorithms for multiplication, and multiplication of an element of any additive group by an integer corresponds to repeated addition; now we consider algorithms for computing powers x^n, and an integer power n of any element x of a multiplicative group corresponds to repeated multiplication. In fact, I will consider only $n \in \mathbb{N}$, in which case x^n can be defined in any (multiplicative) monoid. For $n \in \mathbb{Z}^+$ it would suffice for x to be a member of a semigroup, and the multiplication could be replaced by *any* associative binary operation.

The naïve algorithm to compute x^n, $n \in \mathbb{N}$, simply multiplies 1 by x n times. (If one were to multiply x by itself $n-1$ times then $n = 0$ would become a special case, and it is best to avoid special cases if possible, because the resulting algorithm is simpler, more elegant, and therefore more likely to be correct!) Hence the naïve algorithm looks like this:

input: $x \in$ some multiplicative monoid, $n \in \mathbb{N}$
$z := 1$; {the result}

for $i := 1$ to n do $z := xz$.
output: $z = x^n$

This algorithm requires n multiplications, of which $n - 1$ are non-trivial.

An alternative approach is to think recursively; express the solution to the problem in terms of a simpler version of the same problem. If n is even then $x^n = (x^{n/2})^2$, and if n is odd then $x^n = x(x^{(n-1)/2})^2$. A recursive formulation needs a base case that can be solved non-recursively (in order to stop the recursion), and in this case it is that $x^0 = 1$. This leads immediately to the following recursive algorithm:

input: $x \in$ some multiplicative monoid, $n \in \mathbb{N}$
procedure power(x, n);
if $n = 0$ then 1
else if n even then
power$(x, n$ div $2)$
else
power$(x, (n - 1)$ div $2) \times x$.
output: power$(x, n) = x^n$

The successive halving of the exponent used in this algorithm corresponds to its representation in binary. More precisely, if

$$n = \sum_{i=0}^{k-1} e_i 2^i, \quad e_i \in \{0, 1\}, \quad e_{k-1} = 1$$

then

$$x^n = x^{\sum_{i=0}^{k-1} e_i 2^i} = \prod_{i=0}^{k-1} x^{e_i 2^i} = \prod_{i, e_i = 1} x^{2^i}.$$

This formulation suggests the following iterative algorithm, which computes successive even powers of x, and multiplies them into the result when they occur:

input: $x \in$ some multiplicative monoid, $n \in \mathbb{N}$
$y := x$; {successively $y = x, x^2, x^4, \ldots$}
$z := 1$; {the result}
while $n > 0$ do
begin
$m := n$ div 2;
if $n > 2m$ then $z := zy$; {bit was 1}

$y := y^2$;
$n := m$
end.
output: $z = x^n$

To check that this algorithm is correct, convince yourself that it works correctly for $n = 0, 1, 2, 3, 4, 5$, in which case it is probably correct generally. (But this is not a proof!)

The loop in the above algorithm is controlled by successively halving n, and hence it is executed once for each of the k digits of n in *binary* representation. Within the loop there is one multiplication to square y, and one multiplication when the current bit of n is 1, giving the total number of multiplications N to be

$$N = k + \sum_{i=0}^{k-1} e_i \;\Rightarrow\; k < N \leq 2k$$

(since at least the highest-order bit of n must be 1). If n has k binary digits as above then $2^{k-1} \leq n < 2^k$, and hence $k = 1 + \lfloor \lg n \rfloor$, so $\lfloor \lg n \rfloor \leq N \leq 2(1 + \lfloor \lg n \rfloor)$. In fact, 1 multiplication in the naïve algorithm is trivial (a multiplication by 1), as is the case in the binary algorithm, and the last squaring of y is unnecessary and could be avoided by taking slightly more care. Therefore, the multiplication counts N for the two algorithms are really $N = n - 1$ and $\lfloor \lg n \rfloor \leq N \leq 2\lfloor \lg n \rfloor$ respectively, e.g. if $n = 10^6$ then $N = 999,999$ and $N \leq 38$.

The recursive binary algorithm involves precisely the same operations as the iterative binary algorithm, and so must have exactly the same multiplication count.

6.1 Full complexity of power computation

We have bounded the number of multiplications required, but this may not be a true indication of the complexity, because we also need to consider the complexity of each multiplication operation itself. If, for example, x is an element of a finite monoid then the complexity of each multiplication is bounded by a constant, so the multiplication count is a fair measure of the overall complexity.

However, at present we are more interested in the case that x is a long integer, having (say) h digits in base-B representation, and let us suppose that the complexity of multiplying integers having m and m' digits is cmm',

i.e. we are using a direct multiplication algorithm. In the naïve power algorithm the length of one of the multiplicands increases linearly, giving the complexity in terms of small-integer operations to be

$$C_1(n) = ch(h + 2h + 3h + \cdots + (n-2)h) \approx \frac{1}{2}ch^2n^2$$

by summing the geometric series.

In the binary algorithm (avoiding unnecessary multiplications), initially y and z are both h-digit integers. In the worst case, in which every bit of n is 1, both multiplications are performed on each of the $k-1$ executions of the loop, so the lengths of y and z double, and there is one multiplication of the final values of y and z. Then an upper bound on the true complexity is given by

$$\begin{aligned} C_2(n) &= 2ch^2(1 + 2^2 + (2^2)^2 + (2^3)^2 + \cdots + (2^{k-2})^2) + ch^2(2^{k-1})^2 \\ &= ch^2\left(2\sum_{i=0}^{k-2} 2^{2i} + 2^{2(k-1)}\right) \\ &\approx ch^2\left(\frac{2^{2k}}{6} + \frac{2^{2k}}{4}\right) = \frac{5}{12}ch^2 2^{2k}. \end{aligned}$$

To compare the two complexities, put $n \approx 2^k$ to give

$$C_2(n) \approx \frac{5}{12}ch^2n^2,$$

which is essentially the *same* as $C_1(n)$. A more careful analysis of the binary algorithm would show that its complexity varies around that of the naïve algorithm, but always within a small factor.

Hence we have the somewhat counter-intuitive result that merely reducing the number of multiplications performed on long integers does not necessarily reduce the overall complexity of an algorithm, because one multiplication of two very long integers may be significantly more complex than many multiplications of shorter integers. By balancing the sizes of the integers being multiplied, the binary algorithm effectively makes each of its multiplications more complex.

6.2 Application to linear recurrence relations

An h-step linear recurrence relation (or difference equation) determining (or satisfied by) a sequence $\{u_r\}$ has the form

$$u_n = a_1u_{n-1} + a_2u_{n-2} + \cdots + a_hu_{n-h}, \quad n \geq h.$$

For example, a discretization of a differential equation for purposes of numerical solution takes this form. If U_n denotes a column vector with elements $\{u_n, u_{n-1}, \ldots, u_{n-h+1}\}$ then the above recurrence relation can be written in the matrix form

$$U_n = AU_{n-1},$$

where

$$A = \begin{pmatrix} a_1 & a_2 & \cdots & a_{h-1} & a_h \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}.$$

Then

$$U_n = A^{n-h+1}U_{h-1},$$

which can be computed using $O(\lg n)$ matrix multiplications.

The binary method is particularly useful when computations are made modulo some integer, which therefore keeps the size of the integers involved bounded.

7 Greatest common divisors

We have seen that a greatest common divisor can be defined in any integral domain, and moreover in a gcd domain it can be expressed as a linear combination in the form

$$\gcd(a, b) = sa + tb.$$

In a Euclidean domain, which is a special case of a gcd domain, the Euclidean division property leads to an algorithm to compute both a gcd and the coefficients in the linear combination. However, a gcd is determined only up to associates. In $\mathbb{Z}$, which is a Euclidean domain if the absolute value function is used as the degree function, associates differ by their sign, and it is conventional to define *the* gcd to be positive.

7.1 Euclid's algorithm

The algorithm to compute gcds is due to Euclid (after whom the domain is named), and has been known for over two thousand years.

Let $a, b \in \mathbb{Z}$ be not both zero (otherwise the gcd is undefined). To construct a gcd algorithm, let us think recursively again. Firstly, note that every integer divides zero, so that

$$\gcd(a, 0) = |a|,$$

which gives us a base case for a recursive definition. The integer division property (a special case of the Euclidean division property) asserts that if $b \neq 0$ then there exist unique $q, r \in \mathbb{Z}$ such that

$$a = bq + r, \quad 0 \leq r < |b|,$$

which we denote by $q = a \operatorname{div} b$, $r = a \bmod b$. Then

$$c \mid a,\ c \mid b \ \Rightarrow\ c \mid r = a - bq$$

and so if $b \neq 0$

$$\gcd(a, b) = \gcd(b, r) = \gcd(b, a \bmod b).$$

Hence we have an evaluation for $\gcd(a, b)$ in both of the cases that $b = 0$ and $b \neq 0$, which suggests the following (recursive) algorithm:

input: $a, b \in \mathbb{Z}$, not both 0
procedure $\gcd(a, b)$;
if $b = 0$ then $|a|$ else $\gcd(b, a \bmod b)$.
output: $\gcd(a, b)$

To show that this algorithm terminates (i.e. stops after a finite number of recursive calls), observe the following. If $|a| \geq |b|$ then $q \neq 0$ and so $|r| < |b|$, hence in the recursive call the first argument does not increase and the second argument strictly decreases towards a minimum value of 0, which it must eventually attain (because it is *strictly* decreasing) in at most $|b|$ recursions. If $|a| < |b|$ then $q = 0$, $r = a$, and the recursive call simply exchanges the arguments. Then $|a| \geq |b|$ in the next recursive call of $\gcd(a, b)$, and $|a| > |b|$ in all subsequent recursive calls, so this exchange operation occurs at most once.

The above algorithm can easily be converted into an iterative algorithm, i.e. one that is not explicitly recursive. To do this, it is convenient to let the initial values of the variables a, b, r, q be p_0, p_1, p_2, q_1, and on each recursive call to increment the subscripts by 1. This notation reflects the assignments

$a := b$, $b := r$ implicit in the recursive calls. Then the sequence of recursive calls is equivalent to the sequence of instances of the division property shown in the left half of the following table:

$\gcd(a,b)$	$= sa + tb$
$p_0 = a, \quad p_1 = b$	$P_0 = A, \quad P_1 = B$
$p_0 = p_1q_1 + p_2$	$P_2 = P_0 - P_1q_1$
$p_1 = p_2q_2 + p_3$	$P_3 = P_1 - P_2q_2$
$\vdots$	$\vdots$
$p_{n-2} = p_{n-1}q_{n-1} + p_n$	$P_n = P_{n-2} - P_{n-1}q_{n-1}$
$p_{n-1} = p_nq_n + 0$	
$\gcd(p_0, p_1) = p_n$	$P_n = sA + tB$

Note that the quotients q_i are not actually used in the computation of the gcd, and they were not used in the above recursive algorithm. I will give an explicit iterative algorithm in the following extended context.

7.2 The extended Euclidean algorithm

The gcd of a, b and the coefficients s, t in the linear combination

$$\gcd(a, b) = sa + tb$$

can be found together by using a small extension of Euclid's algorithm. To see how this works, consider the right half of the above table, in which I have used capital letters for some of the symbols. Otherwise, the equations in the right half are identical to those in the left half, but with the remainders isolated on the left. Clearly, P_2 is a linear combination of $P_0 = A$ and $P_1 = B$, and hence so are $P_3, \ldots, P_n$, and since P_n is the gcd, the coefficients of A, B in the expression that is the value of P_n are the required expansion coefficients s, t. Then suppose that A, B are symbolic objects that are distinct from a, b, that the P_i are variables, distinct from the p_i, that are assigned values according to the right half of the above table, but that the q_i are the values determined by the equations in the left half of the table. This allows us to run the two calculations together. One obvious choice for the symbolic objects to represent A, B is two linearly independent 2-vectors, and if these are chosen to be $A = (1, 0)$, $B = (0, 1)$ then $P_n = sA + tB = (s, t)$. However, other representations are possible, such as keeping A, B as unbound (unassigned) variables.

Based on this analysis, here is an iterative extended Euclidean algorithm:

input: $a, b \in \mathbb{Z}$, not both 0
$p_0 := a$; $P_0 := (1, 0)$;
$p_1 := b$; $P_1 := (0, 1)$;
while $p_1 \neq 0$ do
begin
    $q := p_0$ div p_1;
    $p_2 := p_0$ mod p_1; {or $p_0 - qp_1$}
    $p_0 := p_1$; $p_1 := p_2$;
    $P_2 := P_0 - qP_1$;
    $P_0 := P_1$; $P_1 := P_2$
end;
{Pick the positive gcd:}
if $p_0 < 0$ then
begin
    $p_0 := -p_0$;
    $P_0 := -P_0$
end.
output: $\gcd(a, b) = p_0 = sa + bt$, $(s, t) = P_0$

As we will see in later notes, the main practical use of the extended Euclidean algorithm is to compute inverses in quotient rings and finite fields, and it can also be used to compute partial fraction decompositions, which are useful in integration and summation, for example.

7.3 Complexity of Euclid's algorithm

The argument used above to show termination of the algorithm also shows that the algorithm requires at most $|b|$ iterations to reduce b to zero, plus perhaps 1 iteration to initially exchange a and b, and so the complexity measured in terms of iterations is bounded by $O(N)$ where $N = \min(|a|, |b|)$. However, this assumes that $|b|$ decreases in steps of 1, which is unduly pessimistic. The following better complexity bound is due to G. E. Collins[5] and is described by Lipson.

We may assume that $a \geq b > 0$. Then Euclid's algorithm computes a *remainder sequence* $a \geq b > p_2 > \ldots > p_n > p_{n+1} = 0$ where $p_n = g =$

[5]Section 2 of "Computing Time Analyses for some Arithmetic and Algebraic Algorithms", in R. G. Tobey (ed.), *Proc. 1968 Summer Institute on Symbolic Mathematical Computation*, IBM Programming Laboratory Report FSC69-0312, June 1969.

$\gcd(a,b)$. The remainders satisfy

$$\begin{aligned} p_{i-1} &= p_i q_i + p_{i+1}, \quad (1 \le i \le n-1) \\ &> p_{i+1} q_i + p_{i+1} = p_{i+1}(q_i + 1), \end{aligned}$$

and hence

$$\prod_{i=1}^{n-1} p_{i-1} > \prod_{i=1}^{n-1} p_{i+1}(q_i + 1).$$

Cancelling $\prod_{i=2}^{n-2} p_i > 0$ simplifies this to

$$p_0 p_1 > p_{n-1} p_n \prod_{i=1}^{n-1} (q_i + 1) = p_n^2 q_n \prod_{i=1}^{n-1} (q_i + 1)$$

(because $p_{n+1} = 0 \;\Rightarrow\; p_{n-1} = p_n q_n$). Substituting now $p_0 = a, p_1 = b, p_n = g$ and using $q_i \ge 1$, $1 \le i \le n-1$ (because $p_{i-1} > p_i$), and $q_n \ge 2$ (because $p_{n-1} > p_n$), gives

$$ab > g^2 2.2^{n-1} \ge 2^n$$

(because $g \ge 1$). If $a \ge b$ then $a^2 \ge ab > 2^n$ and hence $n < 2 \lg a$, so we have proved

Theorem 6 *Let $a, b \in \mathbb{N}$, $N \ge a \ge b \ge 0$; then Euclid's algorithm computes $\gcd(a,b)$ in $< 2 \lg N = O(\lg N)$ iterations.*

The tightest possible bound, due to Lamé, of $1.44 \lg N$ is intimately related to Fibonacci numbers.

However, this number of iterations is not a true measure of complexity if long integers are involved. If a long integer N has n digits in some base, then $O(\lg N) = O(n)$ bounds the number of iterations, and each iteration involves a division operation, the complexity of which is certainly bounded by $O(n^2)$, giving an overall complexity of $O(n^3)$. With a bit more care this bound can in fact be reduced to $O(n^2)$ (or in principle a bit less). Nevertheless, gcd computations are quite costly, and so should be avoided if possible!

8 Lowest common multiples

A lowest common multiple (lcm) of the elements $a, b \ne 0$ of an integral domain D is an element $\ell \in D$ such that:

1. $a \mid \ell,\ b \mid \ell$;

2. $a \,|\, c,\ b \,|\, c \ \Rightarrow\ \ell \,|\, c.$

It can be computed as

$$\operatorname{lcm}(a, b) = ab/\gcd(a, b)$$

and a distinguished lcm can be chosen as for gcds.

9 Rational arithmetic

Given the above algorithms for performing long integer arithmetic, it is straightforward to construct algorithms for performing rational arithmetic. The set of rational numbers $\mathbb{Q}$ is the quotient field (field of quotients or fractions) of the integers $\mathbb{Z}$, so we must implement the operations required by the construction of a quotient field. As I discussed in Notes 1, it is convenient to use a canonical representation, and a rational number is formally an equivalence class of pairs of integers $[(a, b)], b \neq 0$, among which we want to choose a distinguished (canonical) representative. Because, as we have just seen, it is advantageous to keep the integers with which we work as small as possible in order to minimize computing times, we choose the canonical rational representation of $[(a, b)]$ to have a, b as small as possible, which means dividing out their gcd. In doing so, we choose the sign of the gcd so as to make the denominator of the canonical representative strictly positive. This gives the following canonicalization algorithm:

$$\frac{a}{b} \to \frac{\operatorname{sign}(b).a/\gcd(a, b)}{|b|/\gcd(a, b)}.$$

All routines that manipulate rational numbers, including input routines, should return canonical representations. One could compute arithmetic operations on rational numbers in the standard way and then canonicalize the result, but this can involve computing gcds of larger numbers than necessary, and slightly less obvious algorithms can be faster. For example, to compute

$$\frac{a}{b} \times \frac{c}{d} = \frac{p}{q}$$

one could compute $p = ac, q = bd$ and then canonicalize, which requires $\gcd(p, q) = \gcd(ac, bd)$. But if the input numbers were already canonical, then a, b are relatively prime and so are c, d, and hence $\gcd(p, q) = \gcd(ac, bd) = \gcd(a, d)\gcd(b, c)$. Therefore, it is better to compute

$$p = \frac{a}{\gcd(a, d)} \cdot \frac{c}{\gcd(b, c)}, \quad q = \frac{b}{\gcd(b, c)} \cdot \frac{d}{\gcd(a, d)}$$

which is then canonical.

Similarly, to compute

$$\frac{a}{b} + \frac{c}{d} = \frac{p}{q}$$

one could compute $p = ad + bc, q = bd$ and then canonicalize, which requires $\gcd(p, q) = \gcd(ad + bc, bd)$. But it is better to use the lcm rather than simply the product of the denominators as the initial denominator of the sum, and so to compute

$$p = a \cdot \frac{d}{\gcd(b, d)} + c \cdot \frac{b}{\gcd(b, d)}, \quad q = \operatorname{lcm}(b, d) = \frac{bd}{\gcd(b, d)},$$

and *then* canonicalize p/q.

10 Exercises

The assessed questions in this set of exercises are the first four.

1. (** **Assessed** **)
Give an example subtraction in which a carry propages through all digit positions, analogous to the additive example given in the notes above.

2. (** **Assessed** **)
Using base-10 representation, run the Euclidean division algorithm in detail by hand for $a = 12345$, $b = 78$.

3. (** **Assessed** **)
Show how the binary method computes the power x^{15}, and in particular show how many multiplications it requires. Devise an algorithm to compute this particular power using fewer multiplications, thereby demonstrating that the power method is not necessarily optimal.

4. (** **Assessed** **)
Run the extended Euclidean gcd algorithm by hand to compute g, s, t in the relation

$$\gcd(286, 91) = g = 286s + 91t.$$

5. Design two recursive algorithms (a) and (b) to perform addition of long integers using a list representation with the digits in decreasing weight order, such that (a) generates the sum list in the natural order,

i.e. by prepending cells representing digits of higher weight to the front of the list, and (b) outputs the digits of the sum in decreasing weight order *without* first constructing it as a list. Try the same for multiplication. Implement them – Lisp or symbolic-mode REDUCE would be convenient languages, otherwise you may also have to write the list processor, which is not trivial!

6. Using base-10 representation, run the two multiplication algorithms by hand for $a = 12345$, $b = 6789$, and compare them with the conventional method of performing long multiplication by hand. Compare the *space* complexities of the methods.

7. Develop explicit algorithms for performing rational arithmetic and returning canonical results. Implement and test them, which is easy in almost any language including FORTRAN, but slightly more elegant in a language that provides structured data types such as Pascal or C.

8. Implement and test some or all of the algorithms discussed in the notes in your favourite programming language – if that happens to be FORTRAN you will have trouble with the recursive algorithms unless you have access to FORTRAN 90. But even in FORTRAN it is fun to write a set of routines to compute exact factorials of large numbers – much larger than supported by the standard number representations. To do that you will need to design and implement I/O routines in addition to those discussed here.