INPUT	OUTPUT
$\frac{495782912874914951515089054258695 78}{7436743693123724272726335813880 4367}$	$\frac{2}{3}$

13.9 Arbitrary-Precision Arithmetic

Input description: Two very large integers, x and y.

Problem description: What is $x + y$, $x - y$, $x \times y$, and x/y?

Discussion: Any programming language rising above basic assembler supports single- and perhaps double-precision integer/real addition, subtraction, multiplication, and division. But what if we wanted to represent the national debt of the United States in pennies? One trillion dollars worth of pennies requires 15 decimal digits, which is far more than can fit into a 32-bit integer.

Other applications require *much* larger integers. The RSA algorithm for public-key cryptography recommends integer keys of at least 1000 digits to achieve adequate security. Experimenting with number-theoretic conjectures for fun or research requires playing with large numbers. I once solved a minor open problem [GKP89] by performing an exact computation on the integer $\binom{5906}{2953} \approx 9.93285 \times 10^{1775}$.

What should you do when you need large integers?

- *Am I solving a problem instance requiring large integers, or do I have an embedded application?* – If you just need the answer to a specific problem with large integers, such as in the number theory application above, you should consider using a computer algebra system like Maple or Mathematica. These provide arbitrary-precision arithmetic as a default and use nice Lisp-like programming languages as a front end—together often reducing your problem to a 5- to 10- line program.

  If you have an embedded application requiring high-precision arithmetic instead, you should use an existing arbitrary precision math library. You are likely to get additional functions beyond the four basic operations for computing things like greatest common divisor in the bargain. See the Implementations section for details.

- *Do I need high- or arbitrary-precision arithmetic?* – Is there an upper bound on how big your integers can get, or do you really need *arbitrary*-precision—i.e. , unbounded. This determines whether you can use a fixed-length array to represent your integers as opposed to a linked-list of digits. The array is likely to be simpler and will not prove a constraint in most applications.

- *What base should I do arithmetic in?* – It is perhaps simplest to implement your own high-precision arithmetic package in decimal, and thus represent each integer as a string of base-10 digits. However, it is far more efficient to use a higher base, ideally equal to the square root of the largest integer supported fully by hardware arithmetic.

  Why? The higher the base, the fewer digits we need to represent a number. Compare 64 decimal with 1000000 binary. Since hardware addition usually takes one clock cycle independent of value of the actual numbers, best performance is achieved using the highest supported base. The factor limiting us to base $b = \sqrt{maxint}$ is the desire to avoid overflow when multiplying two of these "digits" together.

  The primary complication of using a larger base is that integers must be converted to and from base-10 for input and output. The conversion is easily performed once all four high-precision arithmetical operations are supported.

- *How low-level are you willing to get for fast computation?* – Hardware addition is much faster than a subroutine call, so you take a significant hit on speed using high-precision arithmetic when low-precision arithmetic suffices. High-precision arithmetic is one of few problems in this book where inner loops in assembly language prove the right idea to speed things up. Similarly, using bit-level masking and shift operations instead of arithmetical operations can be a win if you really understand the machine integer representation.

The algorithm of choice for each of the five basic arithmetic operations is as follows:

- *Addition* – The basic schoolhouse method of lining up the decimal points and then adding the digits from right to left with "carries" runs to time linear in the number of digits. More sophisticated carry-look-ahead parallel algorithms are available for low-level hardware implementation. They are presumably used on your microprocessor for low-precision addition.

- *Subtraction* – By fooling with the sign bits of the numbers, subtraction can be a special considered case of addition: $(A - (-B)) = (A + B)$. The tricky part of subtraction is performing the "borrow." This can be simplified by always subtracting from the number with the larger absolute value and adjusting the signs afterwards, so we can be certain there will always be something to borrow from.

- *Multiplication* – Repeated addition will take exponential time on large integers, so stay away from it. The digit-by-digit schoolhouse method is reasonable to program and will work much better, presumably well enough for your application. On very large integers, Karatsuba's $O(n^{1.59})$ divide-and-conquer algorithm wins. Dan Grayson, author of Mathematica's arbitrary-precision arithmetic, found that the switch-over happened at well under 100 digits.

Even faster for very large integers is an algorithm based on Fourier transforms. Such algorithms are discussed in Section 13.11 (page 431).

- *Division* – Repeated subtraction will take exponential time, so the easiest reasonable algorithm to use is the long-division method you hated in school. This is fairly complicated, requiring arbitrary-precision multiplication and subtraction as subroutines, as well as trial-and-error, to determine the correct digit at each position of the quotient.

  In fact, integer division can be reduced to integer multiplication, although in a nontrivial way, so if you are implementing asymptotically fast multiplication, you can reuse that effort in long division. See the references below for details.

- *Exponentiation* – We can compute a^n using $n-1$ multiplications, by computing $a \times a \times \ldots \times a$. However, a much better divide-and-conquer algorithm is based on the observation that $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$. If n is even, then $a^n = (a^{n/2})^2$. If n is odd, then $a^n = a(a^{\lfloor n/2 \rfloor})^2$. In either case, we have halved the size of our exponent at the cost of at most two multiplications, so $O(\lg n)$ multiplications suffice to compute the final value:

```
function power(a, n)
    if (n = 0) return(1)
    x = power(a, ⌊n/2⌋)
    if (n is even) then return(x^2)
        else return(a × x^2)
```

High- but not arbitrary-precision arithmetic can be conveniently performed using the Chinese remainder theorem and modular arithmetic. The *Chinese remainder theorem* states that an integer between 1 and $P = \prod_{i=1}^{k} p_i$ is uniquely determined by its set of residues mod p_i, where each p_i, p_j are relatively prime integers. Addition, subtraction, and multiplication (but not division) can be supported using such residue systems, with the advantage that large integers can be manipulated without complicated data structures.

Many of these algorithms for computations on long integers can be directly applied to computations on polynomials. See the references for more details. A particularly useful algorithm is Horner's rule for fast polynomial evaluation. When $P(x) = \sum_{i=0}^{n} c_i \cdot x^i$ is blindly evaluated term by term, $O(n^2)$ multiplications will be performed. Much better is observing that $P(x) = c_0 + x(c_1 + x(c_2 + x(c_3 + \ldots)))$, the evaluation of which uses only a linear number of operations.

Implementations: All major commercial computer algebra systems incorporate high-precision arithmetic, including Maple, Mathematica, Axiom, and Macsyma. If you have access to one of these, this is your best option for a quick, nonembedded

application. The rest of this section focuses on source code available for embedded applications.

The premier C/C++ library for fast, arbitrary-precision is the GNU Multiple Precision Arithmetic Library (GMP), which operates on signed integers, rational numbers, and floating point numbers. It is widely used and well-supported, and available at *http://gmplib.org/*.

The `java.math BigInteger` class provides arbitrary-precision analogues to all of Java's primitive integer operators. `BigInteger` provides additional operations for modular arithmetic, GCD calculation, primality testing, prime generation, bit manipulation, and a few other miscellaneous operations.

A lower-performance, less-tested, but more personal implementation of high-precision arithmetic appears in the library from my book *Programming Challenges* [SR03]. See Section 19.1.10 (page 661) for details.

Several general systems for computational number theory are available. Each of these supports operations of arbitrary-precision integers. Information about the PARI, LiDIA, NTL and MIRACL number-theoretic libraries can be found in Section 13.8 (page 420).

ARPREC is a C++/Fortran-90 arbitrary precision package with an associated interactive calculator. MPFUN90 is a similar package written exclusively in Fortran-90. Both are available at *http://crd.lbl.gov/∼dhbailey/mpdist/*. Algorithm 693 [Smi91] of the *Collected Algorithms of the ACM* is a Fortran implementation of floating-point, multiple-precision arithmetic. See Section 19.1.5 (page 659).

Notes: Knuth [Knu97b] is the primary reference on algorithms for all basic arithmetic operations, including implementations of them in the MIX assembly language. Bach and Shallit [BS96] and Shoup [Sho05] provide more recent treatments of computational number theory.

Expositions on the $O(n^{1.59})$-time divide-and-conquer algorithm for multiplication [KO63] include [AHU74, Man89]. An FFT-based algorithm multiplies two n-bit numbers in $O(n \lg n \lg \lg n)$ time and is due to Schönhage and Strassen [SS71]. Expositions include [AHU74, Knu97b]. The reduction between integer division and multiplication is presented in [AHU74, Knu97b]. Applications of fast multiplication to other arithmetic operations are presented by Bernstein [Ber04b]

Good expositions of algorithms for modular arithmetic and the Chinese remainder theorem include [AHU74, CLRS01]. A good exposition of circuit-level algorithms for elementary arithmetic algorithms is [CLRS01].

Euclid's algorithm for computing the greatest common divisor of two numbers is perhaps the oldest interesting algorithm. Expositions include [CLRS01, Man89].

Related Problems: Factoring integers (see page 420), cryptography (see page 641).