Arbitrary-precision arithmetic

In computer science, **arbitrary-precision arithmetic**, also called **bignum arithmetic**, **multiple-precision arithmetic**, or sometimes **infinite-precision arithmetic**, indicates that calculations are performed on numbers whose digits of precision are potentially limited only by the available memory of the host system. This contrasts with the faster fixed-precision arithmetic found in most arithmetic logic unit (ALU) hardware, which typically offers between 8 and 64 bits of precision.

Several modern programming languages have built-in support for bignums,[1][2][3][4] and others have libraries available for arbitrary-precision integer and floating-point math. Rather than storing values as a fixed number of bits related to the size of the processor register, these implementations typically use variable-length arrays of digits.

Arbitrary precision is used in applications where the speed of arithmetic is not a limiting factor, or where precise results with very large numbers are required. It should not be confused with the symbolic computation provided by many computer algebra systems, which represent numbers by expressions such as $\pi \cdot \sin(2)$, and can thus *represent* any computable number with infinite precision.

Applications

A common application is public-key cryptography, whose algorithms commonly employ arithmetic with integers having hundreds of digits.[5][6] Another is in situations where artificial limits and overflows would be inappropriate. It is also useful for checking the results of fixed-precision calculations, and for determining optimal or near-optimal values for coefficients needed in formulae, for example the $\sqrt{\frac{1}{3}}$ that appears in Gaussian integration.[7]

Arbitrary precision arithmetic is also used to compute fundamental mathematical constants such as π to millions or more digits and to analyze the properties of the digit strings[8] or more generally to investigate the precise behaviour of functions such as the Riemann zeta function where certain questions are difficult to explore via analytical methods. Another example is in rendering fractal images with an extremely high magnification, such as those found in the Mandelbrot set.

Arbitrary-precision arithmetic can also be used to avoid overflow, which is an inherent limitation of fixed-precision arithmetic. Similar to a five-digit odometer's display which changes from 99999 to 00000, a fixed-precision integer may exhibit *wraparound* if numbers grow too large to represent at the fixed level of precision. Some processors can instead deal with overflow by *saturation,* which means that if a result would be unrepresentable, it is replaced with the nearest representable value. (With 16-bit unsigned saturation, adding any positive amount to 65535 would yield 65535.) Some processors can generate an exception if an arithmetic result exceeds the available precision. Where necessary, the exception can be caught and recovered from—for instance, the operation could be restarted in software using arbitrary-precision arithmetic.

In many cases, the task or the programmer can guarantee that the integer values in a specific application will not grow large enough to cause an overflow. Such guarantees may be based on pragmatic limits: a school attendance program may have a task limit of 4,000 students. A programmer may design the computation so that intermediate results stay within specified precision boundaries.

Some programming languages such as Lisp, Python, Perl, Haskell, Ruby and Raku use, or have an option to use, arbitrary-precision numbers for *all* integer arithmetic. Although this reduces performance, it eliminates the possibility of incorrect results (or exceptions) due to simple overflow. It also makes it possible to guarantee that arithmetic results will be the same on all machines, regardless of any particular machine's word size. The exclusive use of arbitrary-precision numbers in a programming language also simplifies the language, because *a number is a number* and there is no need for multiple types to represent different levels of precision.

Implementation issues

Arbitrary-precision arithmetic is considerably slower than arithmetic using numbers that fit entirely within processor registers, since the latter are usually implemented in hardware arithmetic whereas the former must be implemented in software. Even if the computer lacks hardware for certain operations (such as integer division, or all floating-point operations) and software is provided instead, it will use number sizes closely related to the available hardware registers: one or two words only. There are exceptions, as certain *variable word length* machines of the 1950s and 1960s, notably the IBM 1620, IBM 1401 and the Honeywell 200 series, could manipulate numbers bound only by available storage, with an extra bit that delimited the value.

Numbers can be stored in a fixed-point format, or in a floating-point format as a significand multiplied by an arbitrary exponent. However, since division almost immediately introduces infinitely repeating sequences of digits (such as 4/7 in decimal, or 1/10 in binary), should this possibility arise then either the representation would be truncated at some satisfactory size or else rational numbers would be used: a large integer for the numerator and for the denominator. But even with the greatest common divisor divided out, arithmetic with rational numbers can become unwieldy very quickly: 1/99 − 1/100 = 1/9900, and if 1/101 is then added, the result is 10001/999900.

The size of arbitrary-precision numbers is limited in practice by the total storage available, and computation time.

Numerous algorithms have been developed to efficiently perform arithmetic operations on numbers stored with arbitrary precision. In particular, supposing that N digits are employed, algorithms have been designed to minimize the asymptotic complexity for large N.

The simplest algorithms are for addition and subtraction, where one simply adds or subtracts the digits in sequence, carrying as necessary, which yields an $O(N)$ algorithm (see big O notation).

Comparison is also very simple. Compare the high-order digits (or machine words) until a difference is found. Comparing the rest of the digits/words is not necessary. The worst case is $\Theta(N)$, but usually it will go much faster.

For multiplication, the most straightforward algorithms used for multiplying numbers by hand (as taught in primary school) require $\Theta(N^2)$ operations, but multiplication algorithms that achieve $O(N \log(N) \log(\log(N)))$ complexity have been devised, such as the Schönhage–Strassen algorithm, based on fast Fourier transforms, and there are also algorithms with slightly worse complexity but with sometimes superior real-world performance for smaller N. The Karatsuba multiplication is such an algorithm.

For division, see division algorithm.

For a list of algorithms along with complexity estimates, see computational complexity of mathematical operations.

For examples in x86 assembly, see external links.

Pre-set precision

In some languages such as REXX, the precision of all calculations must be set before doing a calculation. Other languages, such as Python and Ruby, extend the precision automatically to prevent overflow.

Example

The calculation of factorials can easily produce very large numbers. This is not a problem for their usage in many formulas (such as Taylor series) because they appear along with other terms, so that—given careful attention to the order of evaluation—intermediate calculation values are not troublesome. If approximate values of factorial numbers are desired, Stirling's approximation gives good results using floating-point arithmetic. The largest representable value for a fixed-size integer variable may be exceeded even for relatively small arguments as shown in the table below. Even floating-point numbers are soon outranged, so it may help to recast the calculations in terms of the logarithm of the number.

But if exact values for large factorials are desired, then special software is required, as in the pseudocode that follows, which implements the classic algorithm to calculate 1, 1×2, 1×2×3, 1×2×3×4, etc. the successive factorial numbers.

```
constants:
  Limit = 1000                    % Sufficient digits.
  Base = 10                       % The base of the simulated arithmetic.
  FactorialLimit = 365            % Target number to solve, 365!
  tdigit: Array[0:9] of character = ["0","1","2","3","4","5","6","7","8","9"]

variables:
  digit: Array[1:Limit] of 0..9   % The big number.
  carry, d: Integer               % Assistants during multiplication.
  last: Integer                   % Index into the big number's digits.
  text: Array[1:Limit] of character % Scratchpad for the output.

digit[*] := 0                     % Clear the whole array.
last := 1                         % The big number starts as a single-digit,
digit[1] := 1                     % its only digit is 1.

for n := 1 to FactorialLimit:     % Step through producing 1!, 2!, 3!, 4!, etc.

  carry := 0                      % Start a multiply by n.
  for i := 1 to last:             % Step along every digit.
    d := digit[i] * n + carry     % Multiply a single digit.
    digit[i] := d mod Base        % Keep the low-order digit of the result.
    carry := d div Base           % Carry over to the next digit.

  while carry > 0:                % Store the remaining carry in the big number.
    if last >= Limit: error("overflow")
    last := last + 1              % One more digit.
    digit[last] := carry mod Base
    carry := carry div Base       % Strip the last digit off the carry.

  text[*] := " "                  % Now prepare the output.
  for i := 1 to last:             % Translate from binary to text.
    text[Limit - i + 1] := tdigit[digit[i]]  % Reversing the order.
  print text[Limit - last + 1:Limit], " = ", n, "!"
```

With the example in view, a number of details can be discussed. The most important is the choice of the representation of the big number. In this case, only integer values are required for digits, so an array of fixed-width integers is adequate. It is convenient to have successive elements of the array represent higher powers of the base.

The second most important decision is in the choice of the base of arithmetic, here ten. There are many considerations. The scratchpad variable *d* must be able to hold the result of a single-digit multiply *plus the carry* from the prior digit's multiply. In base ten, a sixteen-bit integer is certainly adequate as it allows up to 32767. However, this example cheats, in that the value of *n* is not itself limited to a single digit. This has the consequence that the method will fail for $n > 3200$ or so. In a more general implementation, *n* would also use a multi-digit representation. A second consequence of the shortcut is that after the multi-digit multiply has been completed, the last value of *carry* may need to be carried into multiple higher-order digits, not just one.

There is also the issue of printing the result in base ten, for human consideration. Because the base is already ten, the result could be shown simply by printing the successive digits of array *digit*, but they would appear with the highest-order digit last (so that 123 would appear as "321"). The whole array could be printed in reverse order, but that would present the number with leading zeroes ("00000...000123") which may not be appreciated, so this implementation builds the representation in a space-padded text variable and then prints that. The first few results (with spacing every fifth digit and annotation added here) are:

Factorial numbers		Reach of computer integers	
1 =	1!		
2 =	2!		
6 =	3!		
24 =	4!		
120 =	5!	8-bit	255
720 =	6!		
5040 =	7!		
40320 =	8!	16-bit	65535
3 62880 =	9!		
36 28800 =	10!		
399 16800 =	11!		
4790 01600 =	12!	32-bit	42949 67295
62270 20800 =	13!		
8 71782 91200 =	14!		
130 76743 68000 =	15!		
2092 27898 88000 =	16!		
35568 74280 96000 =	17!		
6 40237 37057 28000 =	18!		
121 64510 04088 32000 =	19!		
2432 90200 81766 40000 =	20!	64-bit	18446 74407 37095 51615
51090 94217 17094 40000 =	21!		
11 24000 72777 76076 80000 =	22!		
258 52016 73888 49766 40000 =	23!		
6204 48401 73323 94393 60000 =	24!		
1 55112 10043 33098 59840 00000 =	25!		
40 32914 61126 60563 55840 00000 =	26!		
1088 88694 50418 35216 07680 00000 =	27!		
30488 83446 11713 86050 15040 00000 =	28!		
8 84176 19937 39701 95454 36160 00000 =	29!		
265 25285 98121 91058 63630 84800 00000 =	30!		
8222 83865 41779 22817 72556 28800 00000 =	31!		
2 63130 83693 36935 30167 21801 21600 00000 =	32!		
86 83317 61881 18864 95518 19440 12800 00000 =	33!		
2952 32799 03960 41408 47618 60964 35200 00000 =	34!	128-bit	3402 82366 92093 84634 63374 60743 17682 11455
1 03331 47966 38614 49296 66651 33752 32000 00000 =	35!		

This implementation could make more effective use of the computer's built in arithmetic. A simple escalation would be to use base 100 (with corresponding changes to the translation process for output), or, with sufficiently wide computer variables (such as 32-bit integers) we could use larger bases, such as 10,000. Working in a power-of-2 base closer to the computer's built-in integer operations offers advantages, although conversion to a decimal base for output becomes more difficult. On typical modern computers, additions and multiplications take constant time independent of the values of the operands (so long as the operands fit in single machine words), so there are large gains in packing as much of a bignumber as possible into each element of the digit array. The computer may also offer facilities for splitting a product into a digit and carry without requiring the two operations of *mod* and *div* as in the example, and nearly all arithmetic units provide a *carry flag* which can be exploited in multiple-precision addition and subtraction. This sort of detail is the

grist of machine-code programmers, and a suitable assembly-language bignumber routine can run faster than the result of the compilation of a high-level language, which does not provide direct access to such facilities but instead maps the high-level statements to its model of the target machine using an optimizing compiler.

For a single-digit multiply the working variables must be able to hold the value $(\text{base}-1)^2$ + carry, where the maximum value of the carry is (base−1). Similarly, the variables used to index the digit array are themselves limited in width. A simple way to extend the indices would be to deal with the bignumber's digits in blocks of some convenient size so that the addressing would be via (block *i*, digit *j*) where *i* and *j* would be small integers, or, one could escalate to employing bignumber techniques for the indexing variables. Ultimately, machine storage capacity and execution time impose limits on the problem size.

History

IBM's first business computer, the IBM 702 (a vacuum-tube machine) of the mid-1950s, implemented integer arithmetic *entirely in hardware* on digit strings of any length from 1 to 511 digits. The earliest widespread software implementation of arbitrary-precision arithmetic was probably that in Maclisp. Later, around 1980, the operating systems VAX/VMS and VM/CMS offered bignum facilities as a collection of string functions in the one case and in the languages EXEC 2 and REXX in the other.

An early widespread implementation was available via the IBM 1620 of 1959–1970. The 1620 was a decimal-digit machine which used discrete transistors, yet it had hardware (that used lookup tables) to perform integer arithmetic on digit strings of a length that could be from two to whatever memory was available. For floating-point arithmetic, the mantissa was restricted to a hundred digits or fewer, and the exponent was restricted to two digits only. The largest memory supplied offered 60 000 digits, however Fortran compilers for the 1620 settled on fixed sizes such as 10, though it could be specified on a control card if the default was not satisfactory.

Software libraries

Arbitrary-precision arithmetic in most computer software is implemented by calling an external library that provides data types and subroutines to store numbers with the requested precision and to perform computations.

Different libraries have different ways of representing arbitrary-precision numbers, some libraries work only with integer numbers, others store floating point numbers in a variety of bases (decimal or binary powers). Rather than representing a number as single value, some store numbers as a numerator/denominator pair (rationals) and some can fully represent computable numbers, though only up to some storage limit. Fundamentally, Turing machines cannot represent all real numbers, as the cardinality of $\mathbb{R}$ exceeds the cardinality of $\mathbb{Z}$.

See also

- Fürer's algorithm
- Karatsuba algorithm
- Mixed-precision arithmetic
- Schönhage–Strassen algorithm
- Toom–Cook multiplication
- Little Endian Base 128

References

1. dotnet-bot. "BigInteger Struct (System.Numerics)" (https://docs.microsoft.com/en-us/dotnet/api/system.numerics.biginteger). *docs.microsoft.com*. Retrieved 2022-02-22.
2. "PEP 237 -- Unifying Long Integers and Integers" (https://peps.python.org/pep-0237/). *Python.org*. Retrieved 2022-05-23.
3. "BigInteger (Java Platform SE 7)" (https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html). *docs.oracle.com*. Retrieved 2022-02-22.
4. "BigInt - JavaScript | MDN" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt). *developer.mozilla.org*. Retrieved 2022-02-22.
5. Jacqui Cheng (May 23, 2007). "Researchers: 307-digit key crack endangers 1024-bit RSA" (https://arstechnica.com/news.ars/post/20070523-researchers-307-digit-key-crack-endangers-1024-bit-rsa.html).
6. "RSA Laboratories - 3.1.5 How large a key should be used in the RSA cryptosystem?" (https://web.archive.org/web/20120401144624/http://www.rsa.com/rsalabs/node.asp?id=2218). Archived from the original (http://www.rsa.com/rsalabs/node.asp?id%3D2218) on 2012-04-01. Retrieved 2012-03-31. recommends important RSA keys be 2048 bits (roughly 600 digits).
7. Laurent Fousse (2006). Intégration numérique avec erreur bornée en précision arbitraire. Modélisation et simulation (https://tel.archives-ouvertes.fr/tel-00477243/en) (Report) (in French). Université Henri Poincaré - Nancy I.
8. R. K. Pathria (1962). "A Statistical Study of the Randomness Among the First 10,000 Digits of Pi" (https://www.ams.org/journals/mcom/1962-16-078/S0025-5718-1962-0144443-7/). *Mathematics of Computation*. **16** (78): 188–197. doi:10.1090/s0025-5718-1962-0144443-7 (https://doi.org/10.1090%2Fs0025-5718-1962-0144443-7). Retrieved 2014-01-10. A quote example from this article: "Such an extreme pattern is dangerous even if diluted by one of its neighbouring blocks"; this was the occurrence of the sequence 77 twenty-eight times in one block of a thousand digits.

Further reading

- Knuth, Donald (2008). *Seminumerical Algorithms*. The Art of Computer Programming. Vol. 2 (3rd ed.). Addison-Wesley. ISBN 978-0-201-89684-8., Section 4.3.1: The Classical Algorithms
- Derick Wood (1984). *Paradigms and Programming with Pascal*. Computer Science Press. ISBN 0-914894-45-5.
- Richard Crandall, Carl Pomerance (2005). *Prime Numbers*. Springer-Verlag. ISBN 9780387252827., Chapter 9: Fast Algorithms for Large-Integer Arithmetic

External links

- Chapter 9.3 of *The Art of Assembly* (https://web.archive.org/web/20101019002107/http://oopweb.com/Assembly/Documents/ArtOfAssembly/Volume/Chapter_9/CH09-3.html#HEADING3-1) by Randall Hyde discusses multiprecision arithmetic, with examples in x86-assembly.
- Rosetta Code task Arbitrary-precision integers (http://rosettacode.org/wiki/Arbitrary-precision_integers_%28included%29) Case studies in the style in which over 95 programming languages compute the value of 5**4**3**2 using arbitrary precision arithmetic.
