Division algorithm

From Wikipedia, the free encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Deutsch
日本語
中文
Edit links

> *This article is about algorithms for division. For the theorem proving the existence of a unique quotient and remainder, see Euclidean division.*

A **division algorithm** is an algorithm which, given two integers N and D, computes their quotient and/or remainder, the result of division. Some are applied by hand, while others are employed by digital circuit designs and software.

Division algorithms fall into two main categories: slow division and fast division. Slow division algorithms produce one digit of the final quotient per iteration. Examples of slow division include restoring, non-performing restoring, non-restoring, and SRT division. Fast division methods start with a close approximation to the final quotient and produce twice as many digits of the final quotient on each iteration. Newton–Raphson and Goldschmidt fall into this category.

Discussion will refer to the form $N/D = (Q, R)$, where

- N = Numerator (dividend)
- D = Denominator (divisor)

is the input, and

- Q = Quotient
- R = Remainder

is the output.

Contents [hide]

Division by repeated subtraction [edit]

The simplest division algorithm, historically incorporated into a greatest common divisor algorithm presented in Euclid's *Elements*, Book VII, Proposition 1, finds the remainder given two positive integers using only subtractions and comparisons:

```
while  N ≥ D do
  N := N − D
end
return N
```

The proof that the quotient and remainder exist and are unique, described at Euclidean division, gives rise to a complete division algorithm using additions, subtractions, and comparisons:

```
function divide(N, D)
  if D = 0 then error(DivisionByZero) end
  if D < 0 then (Q,R) := divide(N, -D); return (-Q, R) end
  if N < 0 then
    (Q,R) := divide(-N, D)
    if R = 0 then return (-Q, 0)
    else return (-Q - 1, D - R) end
  end
  -- At this point, N ≥ 0 and D > 0
  Q := 0; R := N
  while  R ≥ D do
    Q := Q + 1
    R := R - D
```

```
  end
  return (Q, R)
end
```

This procedure always produces R ≥ 0. Although very simple, it takes Ω(Q) steps, and so is exponentially slower than even slow division algorithms like long division. It is useful if Q is known to be small (being an output-sensitive algorithm), and can serve as an executable specification.

Long division [edit]

Main article: Long division

Long division is the standard algorithm used for pen-and-paper division of multidigit numbers expressed in decimal notation. It shifts gradually from the left to the right end of the dividend, subtracting the largest possible multiple of the divisor at each stage; the multiples become the digits of the quotient, and the final difference is the remainder. When used with a binary radix, it forms the basis for the integer division (unsigned) with remainder algorithm below. Short division is an abbreviated form of long division suitable for one-digit divisors. Chunking (also known as the partial quotients method or the hangman method) is a less-efficient form of long division which may be easier to understand.

Integer division (unsigned) with remainder [edit]

The following algorithm, the binary version of the famous long division, will divide *N* by *D*, placing the quotient in *Q* and the remainder in *R*. All values are treated as unsigned integers.[*citation needed*]

```
if D == 0 then error(DivisionByZeroException) end
Q := 0                  -- initialize quotient and remainder to zero
R := 0
for i = n-1...0 do      -- where n is number of bits in N
  R := R << 1           -- left-shift R by 1 bit
  R(0) := N(i)          -- set the least-significant bit of R equal to bit i of the numerator
  if R >= D then
    R := R - D
    Q(i) := 1
  end
end
```

Example [edit]

If we take $N=1100_2$ (12_{10}) and $D=100_2$ (4_{10})

Step 1: Set R=0 and Q=0
Step 2: Take i=3 (one less than the number of bits in N)
Step 3: R=00 (left shifted by 1)
Step 4: R=01 (setting R(0) to N(i))
Step 5: R<D, so skip statement

Step 2: Set i=2
Step 3: R=010
Step 4: R=011
Step 5: R<D, statement skipped

Step 2: Set i=1
Step 3: R=0110
Step 4: R=0110
Step 5: R>=D, statement entered
Step 5b: R=10 (R-D)
Step 5c: Q=10 (setting Q(i) to 1)

Step 2: Set i=0
Step 3: R=100
Step 4: R=100
Step 5: R>=D, statement entered
Step 5b: R=0 (R-D)
Step 5c: Q=11 (setting Q(i) to 1)

end
$Q=11_2$ (3_{10}) and R=0.

Slow division methods [edit]

Slow division methods are all based on a standard recurrence equation

$$P_{j+1} = R \times P_j - q_{n-(j+1)} \times D,$$

where:

- P_j is *j*-th the partial remainder of the division
- *R* is the radix
- $q_{n-(j+1)}$ is the digit of the quotient in position *n-(j+1)*, where the digit positions are numbered from least-significant 0 to most

significant $n - 1$

- n is number of digits in the quotient
- D is the divisor

Restoring division [edit]

Restoring division operates on fixed-point fractional numbers and depends on the following assumptions:[citation needed]

- $D < N$
- $0 < N,D < 1$.

The quotient digits q are formed from the digit set {0,1}.

The basic algorithm for binary (radix 2) restoring division is:

```
P := N
D := D << n             -- P and D need twice the word width of N and Q
for i = n-1..0 do       -- for example 31..0 for 32 bits
  P := 2P - D           -- trial subtraction from shifted value
  if P >= 0 then
    q(i) := 1           -- result-bit 1
  else
    q(i) := 0           -- result-bit 0
    P := P + D          -- new partial remainder is (restored) shifted value
  end
end

-- Where: N = Numerator, D = Denominator, n = #bits, P = Partial remainder, q(i) = bit #i of quotient
```

The above restoring division algorithm can avoid the restoring step by saving the shifted value $2P$ before the subtraction in an additional register T (i.e., $T = P << 1$) and copying register T to P when the result of the subtraction $2P - D$ is negative.

Non-performing restoring division is similar to restoring division except that the value of `2*P[i]` is saved, so D does not need to be added back in for the case of `TP[i] ≤ 0`.

Non-restoring division [edit]

Non-restoring division uses the digit set {−1,1} for the quotient digits instead of {0,1}. The basic algorithm for binary (radix 2) non-restoring division is:

```
P[0] := N
D := D << n
i := 0
while i < n do
  if P[i] >= 0 then
    q[n-(i+1)] := 1
    P[i+1] := 2*P[i] - D
  else
    q[n-(i+1)] := -1
    P[i+1] := 2*P[i] + D
  end if
  i := i + 1
end while
```

Following this algorithm, the quotient is in a non-standard form consisting of digits of −1 and +1. This form needs to be converted to binary to form the final quotient. Example:

Convert the following quotient to the digit set {0,1}: $Q = 111\bar{1}\bar{1}1\bar{1}$

Steps:

1. Mask the negative term: $N = 00010101$
2. Form the two's complement of N: $\bar{N} = 11101011$
3. Mask the positive term: $P = 11101010$
4. Sum P and $\bar{N}$: $Q = 11010101$

SRT division [edit]

Named for its creators (Sweeney, Robertson, and Tocher), SRT division is a popular method for division in many microprocessor implementations. SRT division is similar to non-restoring division, but it uses a lookup table based on the dividend and the divisor to determine each quotient digit. The Intel Pentium processor's infamous floating-point division bug was caused by an incorrectly coded lookup table. Five of the 1066 entries had been mistakenly omitted.[1]

Fast division methods [edit]

Newton–Raphson division [edit]

Newton–Raphson uses Newton's method to find the reciprocal of D, and multiply that reciprocal by N to find the final quotient Q.

The steps of Newton–Raphson division are:

1. Calculate an estimate X_0 for the reciprocal $1/D$ of the divisor D.

2. Compute successively more accurate estimates $X_1, X_2, \ldots, X_S$ of the reciprocal. This is where one employs the Newton–Raphson method as such.
3. Compute the quotient by multiplying the dividend by the reciprocal of the divisor: $Q = NX_S$.

In order to apply Newton's method to find the reciprocal of D, it is necessary to find a function $f(X)$ which has a zero at $X = 1/D$. The obvious such function is $f(X) = DX - 1$, but the Newton–Raphson iteration for this is unhelpful since it cannot be computed without already knowing the reciprocal of D. Moreover, multiple iterations for refining reciprocal are not possible since higher order derivatives do not exist for $f(X)$. A function which does work is $f(X) = (1/X) - D$, for which the Newton–Raphson iteration gives

$$X_{i+1} = X_i - \frac{f(X_i)}{f'(X_i)} = X_i - \frac{1/X_i - D}{-1/X_i^2} = X_i + X_i(1 - DX_i) = X_i(2 - DX_i),$$

which can be calculated from X_i using only multiplication and subtraction, or using two fused multiply–adds.

From a computation point of view the expressions $X_{i+1} = X_i + X_i(1 - DX_i)$ and $X_{i+1} = X_i(2 - DX_i)$ are not equivalent. To obtain a result with a precision of n bits while making use of the second expression one must compute the product between X_i and $(2 - DX_i)$ with double the required precision (2n bits).[citation needed] In contrast the product between X_i and $(1 - DX_i)$ need only be computed with a precision of n bits.[why?]

If the error is defined as $\epsilon_i = DX_i - 1$, then:

$$\begin{aligned}
\epsilon_{i+1} &= DX_{i+1} - 1 \\
&= D(X_i(2 - DX_i)) - 1 \\
&= 2DX_i - D^2X_i^2 - 1 \\
&= -(D^2X_i^2 - 2DX_i + 1) \\
&= -(DX_i - 1)^2 \\
&= -{\epsilon_i}^2.
\end{aligned}$$

This squaring of the error at each iteration step — the so-called quadratic convergence of Newton–Raphson's method — has the effect that the number of correct digits in the result roughly *doubles for every iteration*, a property that becomes extremely valuable when the numbers involved have many digits (e.g. in the large integer domain). But it also means that the initial convergence of the method can be comparatively slow, especially if the initial estimate X_0 is poorly chosen.

Apply a bit-shift to the divisor *D* to scale it so that 0.5 ≤ *D* ≤ 1 . The same bit-shift should be applied to the numerator *N* so that the quotient does not change. Then one could use a linear approximation in the form

$$X_0 = T_1 + T_2D \approx \frac{1}{D}$$

to initialize Newton–Raphson. To minimize the maximum of the relative error of this approximation on interval $[0.5, 1]$ one should use

$$X_0 = \frac{48}{17} - \frac{32}{17}D.$$

The coefficients of the linear approximation are determined as follows. The relative error is $|(T_1 + T_2D - 1/D)/(1/D)| = |D(T_1 + T_2D) - 1|$. The minimum of the maximum relative error is determined by the Chebyshev equioscillation theorem applied to $F(D) = D(T_1 + T_2D) - 1$. The local extremum of $F(D)$ occurs when $F'(D) = 0$, which has solution $D = -T_1/(2T_2)$. The function at the extremum must be of opposite sign as the function at the endpoints, namely, $F(1/2) = F(1) = -F(-T_1/(2T_2))$. The two equations in the two unknowns have solution $T_1 = 48/17$ and $T_2 = -32/17$, and the maximum relative error is $F(1) = 1/17$. Using this approximation, the relative error of the initial value is less than

$$|\epsilon_0| \leq \frac{1}{17} \approx 0.059.$$

It is possible to generate a polynomial fit of degree larger than 1, computing the coefficients using the Remez algorithm. The trade-off is that the initial guess requires more computational cycles but hopefully in exchange for fewer iterations of Newton–Raphson.

Since for this method the convergence is exactly quadratic, it follows that

$$S = \left\lceil \log_2 \frac{P+1}{\log_2 17} \right\rceil$$

steps is enough to calculate the value up to P binary places. This evaluates to 3 for IEEE single precision and 4 for both double precision and double extended formats.

Pseudocode [edit]

The following computes the quotient of N and D with a precision of P binary places:

```
Express D as M × 2^e where 1 ≤ M < 2 (standard floating point representation)
D' := D / 2^(e+1)   // scale between 0.5 and 1, can be performed with bit shift / exponent subtraction
N' := N / 2^(e+1)
X := 48/17 - 32/17 × D'   // precompute constants with same precision as D
repeat ⌈log2((P+1)/log2(17))⌉ times   // can be precomputed based on fixed P
    X := X + X × (1 - D' × X)
end
```

```
return N' × X
```

For example, for a double-precision floating-point division, this method uses 10 multiplies, 9 adds, and 2 shifts.

Goldschmidt division [edit]

Goldschmidt (after Robert Elliott Goldschmidt)[2] division uses an iterative process of repeatedly multiplying both the dividend and divisor by a common factor F_i, chosen such that the divisor converges to 1. This causes the dividend to converge to the sought quotient Q:

$$Q = \frac{N}{D}\frac{F_1}{F_1}\frac{F_2}{F_2}\frac{F_\ldots}{F_\ldots}.$$

The steps for Goldschmidt division are:

1. Generate an estimate for the multiplication factor F_i .
2. Multiply the dividend and divisor by F_i .
3. If the divisor is sufficiently close to 1, return the dividend, otherwise, loop to step 1.

Assuming N/D has been scaled so that $0 < D < 1$, each F_i is based on D:

$$F_{i+1} = 2 - D_i.$$

Multiplying the dividend and divisor by the factor yields:

$$\frac{N_{i+1}}{D_{i+1}} = \frac{N_i}{D_i}\frac{F_{i+1}}{F_{i+1}}.$$

After a sufficient number k of iterations $Q = N_k$.

The Goldschmidt method is used in AMD Athlon CPUs and later models.[3][4]

Binomial theorem [edit]

The Goldschmidt method can be used with factors that allow simplifications by the binomial theorem. Assuming N/D has been scaled by a power of two such that $D \in (\frac{1}{2}, 1]$. We choose $D = 1 - x$ and $F_i = 1 + x^{2^i}$. This yields

$$\frac{N}{1-x} = \frac{N\cdot(1+x)}{1-x^2} = \frac{N\cdot(1+x)\cdot(1+x^2)}{1-x^4} = \cdots = Q' = \frac{N' = N\cdot(1+x)\cdot(1+x^2)\cdots(1+x^{2^{(n-1)}})}{D' = 1-x^{2^n} \approx 1}$$

.

After n steps $(x \in [0, \frac{1}{2}))$, the denominator $1 - x^{2^n}$ can be rounded to 1 with a relative error

$$\epsilon_n = \frac{Q' - N'}{Q'} = x^{2^n}$$

which is maximum at 2^{-2^n} when $x = \frac{1}{2}$, thus providing a minimum precision of 2^n binary digits.

This algorithm is referred to as the IBM method in.[5]

Large integer methods [edit]

Methods designed for hardware implementation generally do not scale to integers with thousands or millions of decimal digits; these frequently occur, for example, in modular reductions in cryptography. For these large integers, more efficient division algorithms transform the problem to use a small number of multiplications, which can then be done using an asymptotically efficient multiplication algorithm such as the Karatsuba algorithm, Toom–Cook multiplication or the Schönhage–Strassen algorithm. It results that the computational complexity of the division is of the same order (up a multiplicative constant) as that of the multiplication. Examples include reduction to multiplication by Newton's method as described above[6] as well as the slightly faster Barrett reduction algorithm.[7][*verification needed*] Newton's method is particularly efficient in scenarios where one must divide by the same divisor many times, since after the initial Newton inversion only one (truncated) multiplication is needed for each division.

Division by a constant [edit]

The division by a constant D is equivalent to the multiplication by its reciprocal. Since the denominator is constant, so is its reciprocal $(1/D)$. Thus it is possible to compute the value of $(1/D)$ once at compile time, and at run time perform the multiplication $N\cdot(1/D)$ rather than the division N/D. In floating point arithmetic the use of $(1/D)$ presents little problem, but in integer arithmetic the reciprocal will always evaluate to zero (assuming $|D| > 1$).

It is not necessary to use specifically $(1/D)$; any value (X/Y) that reduces to $(1/D)$ may be used. For example, for division by 3, the factors 1/3, 2/6, 3/9, or 194/582 could be used. Consequently, if Y were a power of two so the division step reduces to a fast right bit shift. The effect of calculating N/D as $(N\cdot X)/Y$ replaces a division with a multiply and a shift. Note that the parentheses are important, as $N\cdot(X/Y)$ will evaluate to zero.

However, unless D itself is a power of two, there is no X and Y that satisfies the conditions above. Fortunately, it is not necessary for (X/Y) to be exactly equal to $1/D$, but only that it is "close enough" so that the error introduced by the approximation is in the bits that are discarded by the shift operation.[8]

As a concrete example, for 32 bit unsigned integers, division by 3 can be replaced with a multiply by 2863311531 / 2^{33}, a multiplication by 2863311531 followed by a 33 right bit shift. The value of 2863311531 is calculated as $2^{33} / 3$ then rounded to nearest integer.

In some cases, division by a constant can be accomplished in even less time by converting the "multiply by a constant" into a series of shifts and adds or subtracts.[9] Of particular interest is division by 10, for which the exact quotient is obtained, with remainder if required.[10]

Rounding error [edit]

This section requires expansion.
(September 2012)

Round-off error can be introduced by division operations due to limited precision.

Further information: Floating point

See also [edit]

- Multiplication algorithm
- Pentium FDIV bug

References [edit]

1. ^ "Statistical Analysis of Floating Point Flaw". Intel Corporation. 1994. Retrieved 22 October 2013.
2. ^ Goldschmidt, Robert E. (1964). *Applications of Division by Convergence* (Thesis). MSc dissertation. M.I.T.
3. ^ Oberman, Stuart F. (1999). "Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessor". *Proc. IEEE Symposium on Computer Arithmetic*: 106–115.
4. ^ Soderquist, Peter; Leeser, Miriam (July–August 1997). "Division and Square Root: Choosing the Right Implementation". *IEEE Micro* **17** (4): 56–66.
5. ^ Paul Molitor, "Entwurf digitaler Systeme mit VHDL"
6. ^ Hasselström, Karl (2003). *Fast Division of Large Integers: A Comparison of Algorithms* (PDF) (Master's in Computer Science thesis). Royal Institute of Technology. Retrieved 2011-03-23.
7. ^ Paul Barrett (1987). "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor". *Proceedings on Advances in cryptology—CRYPTO '86*. London, UK: Springer-Verlag. pp. 311–323. ISBN 0-387-18047-8.
8. ^ Granlund, Torbjörn; Montgomery, Peter L. (June 1994). "Division by Invariant Integers using Multiplication". *SIGPLAN Notices* (ACM) **29** (6): 61–72. doi:10.1145/773473.178249.
9. ^ Massmind: "Binary Division by a Constant"
10. ^ Vowels, R. A. (1992). "Division by 10". *Australian Computer Journal* **24** (3): 81–85.

External links [edit]

- Computer Arithmetic Algorithms JavaScript Simulator – contains simulators for many different division algorithms
- Doras, Cory (19 October 2011). "Labor of Division (Episode III): Faster Unsigned Division by Constants" (PDF). ridiculous_fish. (Extends division by constants.)
- http://www.dauniv.ac.in/downloads/CArch_PPTs/CompArchCh03L07IntegerDivision.pdf
- http://www.seas.ucla.edu/~ingrid/ee213a/lectures/division_presentV2.pdf

Categories: Binary arithmetic | Computer arithmetic | Division (mathematics) | Computer arithmetic algorithms