Extended Euclidean algorithm

From Wikipedia, the free encyclopedia

In arithmetic and computer programming, the **extended Euclidean algorithm** is an extension to the Euclidean algorithm, which computes, besides the greatest common divisor of integers *a* and *b*, the coefficients of Bézout's identity, that is integers *x* and *y* such that

$$ax + by = \gcd(a, b).$$

It allows one to compute also, with almost no extra cost, the quotients of *a* and *b* by their greatest common divisor.

Extended Euclidean algorithm also refers to a very similar algorithm for computing the polynomial greatest common divisor and the coefficients of Bézout's identity of two univariate polynomials.

The extended Euclidean algorithm is particularly useful when *a* and *b* are coprime, since *x* is the modular multiplicative inverse of *a* modulo *b*, and *y* is the modular multiplicative inverse of *b* modulo *a*. Similarly, the polynomial extended Euclidean algorithm allows one to compute the multiplicative inverse in algebraic field extensions and, in particular in finite fields of non prime order. It follows that both extended Euclidean algorithms are widely used in cryptography. In particular, the computation of the modular multiplicative inverse is an essential step in RSA public-key encryption method.

Contents [hide]

Description [edit]

The standard Euclidean algorithm proceeds by a succession of Euclidean divisions whose quotients are not used, only the *remainders* are kept. For the extended algorithm, the successive quotients are used. More precisely, the standard Euclidean algorithm with *a* and *b* as input, consists of computing a sequence $q_1, \ldots, q_k$ of quotients and a sequence $r_0, \ldots, r_{k+1}$ of remainders such that

$$\begin{aligned}
&r_0 = a \\
&r_1 = b \\
&\ldots \\
&r_{i+1} = r_{i-1} - q_i r_i \quad \text{and} \quad 0 \le r_{i+1} < |r_i| \\
&\ldots
\end{aligned}$$

It is the main property of Euclidean division that the inequalities on the right define uniquely r_{i+1} from r_{i-1} and r_i.

The computation stops when one reaches a remainder r_{k+1} which is zero; the greatest common divisor is then the last non zero remainder r_k.

The extended Euclidean algorithm proceeds similarly, but adds two other sequences, as follows

$$
\begin{array}{ll}
r_0 = a & r_1 = b \\
s_0 = 1 & s_1 = 0 \\
t_0 = 0 & t_1 = 1 \\
\vdots & \\
r_{i+1} = r_{i-1} - q_i r_i \quad \text{and} \quad 0 \le r_{i+1} < |r_i| & \text{(this defines } q_i) \\
s_{i+1} = s_{i-1} - q_i s_i & \\
t_{i+1} = t_{i-1} - q_i t_i & \\
\vdots &
\end{array}
$$

The computation also stops when $r_{k+1} = 0$ and gives

- r_k is the greatest common divisor of the input $a = r_0$ and $b = r_1$.
- The Bézout coefficients are s_k and t_k, that is $\gcd(a, b) = r_k = as_k + bt_k$
- The quotients of *a* and *b* by their greatest common divisor are given by $s_{k+1} = \pm\frac{b}{\gcd(a,b)}$ and $t_{k+1} = \pm\frac{a}{\gcd(a,b)}$

Moreover, if *a* and *b* are both positive, we have

$$|s_k| < \frac{b}{\gcd(a,b)} \quad \text{and} \quad |t_k| < \frac{a}{\gcd(a,b)}.$$

This means that the pair of Bézout's coefficients provided by the extended Euclidean algorithm is one of the two minimal pairs of Bézout coefficients.

Example [edit]

The following table shows how the extended Euclidean algorithm proceeds with input 240 and 46. The greatest common divisor is the last non zero entry, 2 in the column "remainder". The computation stops at row 6, because the remainder in it is 0. Bézout coefficients appear in the last two entries of the second-to-last row. In fact, it is easy to verify that $-9 \times 240 + 47 \times 46 = 2$. Finally the last two entries 23 and -120 of the last row are, up to the sign, the quotients of the input 46 and 240 by the greatest common divisor 2.

index *i*	quotient q_{i-1}	Remainder r_i	s_i	t_i
0		240	1	0
1		46	0	1
2	240 ÷ 46 = 5	240 − 5 × 46 = 10	1 − 5 × 0 = 1	0 − 5 × 1 = −5
3	46 ÷ 10 = 4	46 − 4 × 10 = 6	0 − 4 × 1 = −4	1 − 4 × −5 = 21
4	10 ÷ 6 = 1	10 − 1 × 6 = 4	1 − 1 × −4 = 5	−5 − 1 × 21 = −26
5	6 ÷ 4 = 1	6 − 1 × 4 = 2	−4 − 1 × 5 = −9	21 − 1 × −26 = 47
6	4 ÷ 2 = 2	4 − 2 × 2 = 0	5 − 2 × -9 = 23	−26 − 2 × 47 = −120

Proof [edit]

As $0 \le r_{i+1} < |r_i|$, the sequence of the r_i is a decreasing sequence nonnegative integers (from *i* = 2 on). Thus it must stop with some $r_{k+1} = 0$. This proves that the algorithm stops eventually.

As $r_{i+1} = r_{i-1} - r_i q_i$, the greatest common divisors are the same for (r_{i-1}, r_i) and (r_i, r_{i+1}). This shows that the greatest common divisor of the input $a = r_0, b = r_1$ is the same as that of $r_k, r_{k+1} = 0$. This proves that r_k is the greatest common divisor of *a* and *b*. (Until this point, the proof is the same as that of the classical Euclidean algorithm.)

As $a = r_0$ and $b = r_1$, we have $as_i + bt_i = r_i$ for *i* = 0 and 1. The relation follows by induction for all $i > 1$:
$$r_{i+1} = r_{i-1} - r_i q_i = (as_{i-1} + bt_{i-1}) - (as_i + bt_i)q_i = (as_{i-1} - as_i q_i) + (bt_{i-1} - bt_i q_i) = as_{i+1} + bt_{i+1}.$$
Thus s_k and t_k are Bézout coefficients.

Let us consider the matrix

$$A_i = \begin{pmatrix} s_{i-1} & s_i \\ t_{i-1} & t_i \end{pmatrix}.$$

The recurrence relation may be rewritten in matrix form

$$A_{i+1} = A_i \cdot \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix}.$$

The matrix A_1 is the identity matrix and its determinant is one. The determinant of the rightmost matrix in the preceding formula is -1. It follows that the determinant of A_i is $(-1)^{i-1}$. In particular, for $i = k + 1$, we have $s_k t_{k+1} - t_k s_{k+1} = (-1)^k$. Viewing this as a Bézout's identity, this shows that s_{k+1} and t_{k+1} are coprime. The relation $as_{k+1} + bt_{k+1} = 0$ that has been proved above and Euclid's lemma shows that s_{k+1} divides *b* and t_{k+1} divides *a*. As they are coprime, they are, up to their sign the quotients of *b* and *a* by their greatest common divisor.

Polynomial extended Euclidean algorithm [edit]

See also: Polynomial greatest common divisor § Bézout's identity and extended GCD algorithm

For univariate polynomials with coefficients in a field, everything works in a similar way, Euclidean division, Bézout's identity and extended Euclidean algorithm. The first difference is that, in the Euclidean division and the algorithm, the inequality $0 \leq r_{i+1} < |r_i|$ has to be replaced by an inequality on the degrees $\deg r_{i+1} < \deg r_i$. Otherwise, everything which precedes in this article remains the same, simply by replacing integers by polynomials.

A second difference lies in the bound on the size of the Bézout coefficients provided by the extended Euclidean algorithm, which is more accurate in the polynomial case, leading to the following theorem.

If a and b are two nonzero polynomials, then the extended Euclidean algorithm produces the unique pair of polynomials (*s*, *t*) *such that*

$$as + bt = \gcd(a, b)$$

and

$$\deg s < \deg b - \deg(\gcd(a,b)), \quad \deg t < \deg a - \deg(\gcd(a,b)).$$

A third difference is that, in the polynomial case, the greatest common divisor is defined only up to the multiplication by a non zero constant. There are several ways to define the greatest common divisor unambiguously.

In mathematics, it is common to require that the greatest common divisor be a monic polynomial. To get this, it suffices to divide every element of the output by the leading coefficient of r_k. This allows that, if *a* and *b* are coprime, one gets 1 in the right-hand side of Bézout's inequality. Otherwise, one may get any non-zero constant. In computer algebra, the polynomials have commonly integers coefficients, and this way of normalizing the greatest common divisor introduces too many fractions to be convenient.

The second way to normalize the greatest common divisor in the case of polynomials with integers coefficients is to divide every output by the content of r_k, to get a primitive greatest common divisor. If the input polynomials are coprime, this normalization provides also a greatest common divisor equal to 1. The drawback of this approach is that a lot of fractions should be computed and simplified during the computation.

A third approach consists in extending the algorithm of subresultant pseudo-remainder sequences in a way that is similar to the extension of the Euclidean algorithm to the extended Euclidean algorithm. This allows that, when starting with polynomials with integer coefficients, all polynomials that are computed have integer coefficients. Moreover, every computed remainder r_i is a subresultant polynomial. In particular, if the input polynomials are coprime, then the Bézout's identity becomes

$$as + bt = \operatorname{Res}(a, b),$$

where $\operatorname{Res}(a,b)$ denotes the resultant of *a* and *b*. In this form of Bézout's identity there is no denominator in the formula. If one divides everything by the resultant one gets the classical Bézout's identity, with an explicit common denominator for the rational numbers that appear in it.

Pseudocode [edit]

To implement the algorithm that is described above, one should first remark that only the two last values of the indexed variables are needed at each step. Thus, for saving memory, each indexed variable must be replaced by only two variables.

For simplicity, the following algorithm (and the other algorithms in this article) uses parallel assignments. In a programming language which does not have this feature, the parallel assignments need to be simulated with an auxiliary variable. For example, the first one,

```
(old_r, r) := (r, old_r - quotient *r)
```

is equivalent to

```
prov := r;
r := old_r - quotient * prov;
old_r := prov;
```

and similarly for the other parallel assignments. This leads to the following code:

```
function extended_gcd(a, b)
    s := 0;    old_s := 1
    t := 1;    old_t := 0
    r := b;    old_r := a
    while r ≠ 0
        quotient := old_r div r
        (old_r, r) := (r, old_r - quotient * r)
        (old_s, s) := (s, old_s - quotient * s)
        (old_t, t) := (t, old_t - quotient * t)
    output "Bézout coefficients:", (old_s, old_t)
    output "greatest common divisor:", old_r
    output "quotients by the gcd:", (t, s)
```

The quotients of *a* and *b* by their greatest common divisor, which are output, may have an incorrect sign. This is easy to correct at the end of the computation, but has not been done here for simplifying the code. Similarly, if either *a* or *b* is zero and the other is negative, the greatest common divisor that is output is negative, and all the signs of the output must be changed.

Simplification of fractions [edit]

A fraction $\frac{a}{b}$ is in canonical simplified form if a and b are coprime and b is positive. This canonical simplified form can be obtained by replacing the three **output** lines of the preceding pseudo code by

```
if s = 0 then output "Division by zero"
if s = 1 then output −t      (Optional line, for avoiding output like −t/1
else if s > 0 then output −t/s
else return t/(−s)
```

The proof of this algorithm relies on the fact that *s* and *t* are two coprime integers such that *as* + *bt* = 0, and thus $\frac{a}{b} = -\frac{t}{s}$. To get the canonical simplified form, it suffices to move the minus sign for having a positive denominator.

If *b* divides *a* evenly, the algorithm executes only one iteration, and we have *s* = 1 at the end of the algorithm. It the only case where the output is an integer.

Computing multiplicative inverses in modular structures [edit]

The extended Euclidean algorithm is the basic tool for computing multiplicative inverses in modular structures, typically the modular integers and the algebraic field extensions. An important instance of the latter case are the finite fields of non-prime order.

Modular integers [edit]

Main article: Modular arithmetic

If n is a positive integer, the ring $\mathbf{Z}/n\mathbf{Z}$ may be identified with the set $\{0, 1, ..., n\text{-}1\}$ of the remainders of Euclidean division by n, the addition and the multiplication consisting in taking the remainder by n of the result of the addition and the multiplication of integers. An element a of $\mathbf{Z}/n\mathbf{Z}$ has a multiplicative inverse (that is, it is a unit) if it is coprime to n. In particular, if n is prime, a has a multiplicative inverse if it is not zero (modulo n). Thus $\mathbf{Z}/n\mathbf{Z}$ is a field if and only if n is prime.

Bézout's identity asserts that a and n are coprime if and only if there exist integers s and t such that

$$ns + at = 1$$

Reducing this identity modulo n gives

$$at = 1 \quad \bmod n.$$

Thus t, or, more exactly, the remainder of the division of t by n, is the multiplicative inverse of a modulo n.

To adapt the extended Euclidean algorithm to this problem, one should remark that the Bézout coefficient of n is not needed, and thus does not need to be computed. Also, for getting a result which is positive and lower than *n*, one may use the fact that the integer *t* provided by the algorithm satisfies $|t| < n$. That is, if $t < 0$, one must add n to it at the end. This results in the pseudocode, in which the input *n* is an integer larger than 1.

```
function inverse(a, n)
```

```
    t := 0;     newt := 1;
    r := n;     newr := a;
    while newr ≠ 0
        quotient := r div newr
        (t, newt) := (newt, t - quotient * newt)
        (r, newr) := (newr, r - quotient * newr)
    if r > 1 then return "a is not invertible"
    if t < 0 then t := t + n
    return t
```

Simple algebraic field extensions [edit]

Extended Euclidean algorithm is also the main tool for computing multiplicative inverses in simple algebraic field extensions. An important case, widely used in cryptography and coding theory is that of finite fields of non-prime order. In fact, if p is a prime number, and $q = p^d$, the field of order q is a simple algebraic extension of the prime field of p elements, generated by a root of an irreducible polynomial of degree d.

A simple algebraic extension L of a field K, generated by the root of an irreducible polynomial p of degree d may be identified to the quotient ring $K[X]/\langle p \rangle$, and its elements are in bijective correspondence with the polynomials of degree less than d. The addition in L is the addition of polynomials. The multiplication in L is the remainder of the Euclidean division by p of the product of polynomials. Thus, to complete the arithmetic in L, it remains only to define how to compute multiplicative inverses. This is done by the extended Euclidean algorithm.

The algorithm is very similar to that provided above for computing the modular multiplicative inverse. There are two main differences: firstly the last but one line is not needed, because the Bézout coefficient that is provided has always a degree less than d. Secondly, the greatest common divisor which is provided, when the input polynomials are coprime, may be any non zero element of K; this Bézout coefficient (a polynomial generally of positive degree) has thus to be multiplied by the inverse of this element of K. In the pseudocode which follows, p is a polynomial of degree greater than one, and a is a polynomial. Moreover, **div** is an auxiliary function that computes the quotient of the Euclidean division.

```
function inverse(a, p)
    t := 0;     newt := 1;
    r := p;     newr := a;
    while newr ≠ 0
        quotient := r div newr
        (r, newr) := (newr, r - quotient * newr)
        (t, newt) := (newt, t - quotient * newt)
    if degree(r) > 0 then
        return "Either p is not irreducible or a is a multiple of p"
    return (1/r) * t
```

Example [edit]

For example, if the polynomial used to define the finite field GF(2^8) is $p = x^8 + x^4 + x^3 + x + 1$, and $a = x^6 + x^4 + x + 1$ is the element whose inverse is desired, then performing the algorithm results in the computation described in the following table. Let us recall that in fields of order 2^n, one has $-z = z$ and $z + z = 0$ for every element z in the field). Note also that 1 being the only nonzero element of GF(2), the adjustment in the last line of the pseudocode is not needed.

step	quotient	r, newr	t, newt
		$p = x^8 + x^4 + x^3 + x + 1$	0
		$a = x^6 + x^4 + x + 1$	1
1	$x^2 + 1$	$x^2 = p - a\,(x^2 + 1)$	$x^2 + 1 = 0 - 1 \times (x^2 + 1)$
2	$x^4 + x^2$	$x + 1 = a - x^2\,(x^4 + x^2)$	$x^6 + x^2 + 1 = 1 - (x^4 + x^2)\,(x^2 + 1)$
3	$x + 1$	$1 = x^2 - (x + 1)\,(x + 1)$	$x^7 + x^6 + x^3 + x = (x^2 + 1) - (x + 1)\,(x^6 + x^2 + 1)$
4	$x + 1$	$0 = (x + 1) - 1 \times (x + 1)$	

Thus, the inverse is $x^7 + x^6 + x^3 + x$, as can be confirmed by multiplying the two elements together, and taking the remainder by p of the result.

The case of more than two numbers [edit]

One can handle the case of more than two numbers iteratively. First we show that $\gcd(a,b,c) = \gcd(\gcd(a,b),c)$. To prove this let $d = \gcd(a,b,c)$. By definition of gcd d is a divisor of a and b. Thus $\gcd(a,b) = kd$ for some k. Similarly d is a divisor of c so $c = jd$ for some j. Let $u = \gcd(k,j)$. By our construction of u, $ud | a,b,c$ but since d is the greatest divisor u is a unit. And since

$ud = \gcd(gcd(a, b), c)$ the result is proven.

So if $na + mb = \gcd(a, b)$ then there are x and y such that $x \gcd(a, b) + yc = \gcd(a, b, c)$ so the final equation will be

$$x(na + mb) + yc = (xn)a + (xm)b + yc = \gcd(a, b, c).$$

So then to apply to n numbers we use induction

$$\gcd(a_1, a_2, \ldots, a_n) = \gcd(a_1, \gcd(a_2, \gcd(a_3, \ldots, \gcd(a_{n-1}, a_n))) \ldots),$$

with the equations following directly.

See also [edit]

- Euclidean domain
- Linear congruence theorem

References [edit]

- Knuth, Donald. *The Art of Computer Programming*. Addison-Wesley. Volume 2, Chapter 4.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Pages 859–861 of section 31.2: Greatest common divisor.

External links [edit]

- Source for the form of the algorithm used to determine the multiplicative inverse in GF(2^8)

The Wikibook *Algorithm Implementation* has a page on the topic of: ***Extended Euclidean algorithm***

v · t · e **Number-theoretic algorithms** [hide]

Primality tests	AKS test · APR test · Baillie–PSW · ECPP test · Elliptic curve · Pocklington · Fermat · Lucas · *Lucas–Lehmer* · *Lucas–Lehmer–Riesel* · *Proth's theorem* · *Pépin's* · Quadratic Frobenius test · Solovay–Strassen · Miller–Rabin
Prime-generating	Sieve of Atkin · Sieve of Eratosthenes · Sieve of Sundaram · Wheel factorization
Integer factorization	Continued fraction (CFRAC) · Dixon's · Lenstra elliptic curve (ECM) · Euler's · Pollard's rho · $p-1$ · $p+1$ · Quadratic sieve (QS) · General number field sieve (GNFS) · *Special number field sieve (SNFS)* · Rational sieve · Fermat's · Shanks' square forms · Trial division · Shor's
Multiplication	Ancient Egyptian · Long · Karatsuba · Toom–Cook · Schönhage–Strassen · Fürer's
Discrete logarithm	Baby-step giant-step · Pollard rho · Pollard kangaroo · Pohlig–Hellman · Index calculus · Function field sieve
Greatest common divisor	Binary · Euclidean · **Extended Euclidean** · Lehmer's
Modular square root	Cipolla · Pocklington's · Tonelli–Shanks
Other algorithms	Chakravala · Cornacchia · Integer relation · Integer square root · Modular exponentiation · Schoof's

Italics indicate that algorithm is for numbers of special forms · Smallcaps indicate a deterministic algorithm

Categories: Number theoretic algorithms | Euclid