



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

Беларуская

Deutsch

فارسی

한국어

Русский

Српски / srpski

Edit links

Create account Log in

Article **Talk**

Read

Edit

View history

Search



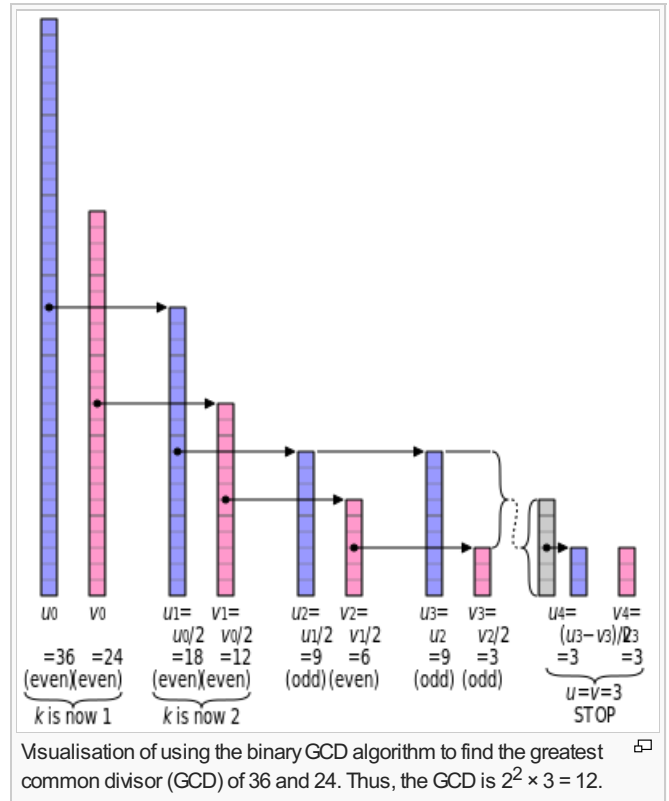
Binary GCD algorithm

From Wikipedia, the free encyclopedia

The **binary GCD algorithm**, also known as **Stein's algorithm**, is an algorithm that computes the **greatest common divisor** of two nonnegative integers. Stein's algorithm uses simpler arithmetic operations than the conventional **Euclidean algorithm**; it replaces division with **arithmetic shifts**, comparisons, and subtraction. Although the algorithm was first published by the Israeli physicist and programmer **Josef Stein** in 1967,^[1] it may have been known in 1st-century China.^[2]

Contents

- Algorithm
- Implementation
 - Recursive version in C
 - Iterative version in C
- Efficiency
- Historical description
- See also
- References
- Further reading
- External links



Algorithm [edit]

The algorithm reduces the problem of finding the GCD by repeatedly applying these identities:

- $\gcd(0, v) = v$, because everything divides zero, and v is the largest number that divides v . Similarly, $\gcd(u, 0) = u$. $\gcd(0, 0)$ is not typically defined, but it is convenient to set $\gcd(0, 0) = 0$.
- If u and v are both even, then $\gcd(u, v) = 2 \cdot \gcd(u/2, v/2)$, because 2 is a common divisor.
- If u is even and v is odd, then $\gcd(u, v) = \gcd(u/2, v)$, because 2 is not a common divisor. Similarly, if u is odd and v is even, then $\gcd(u, v) = \gcd(u, v/2)$.
- If u and v are both odd, and $u \geq v$, then $\gcd(u, v) = \gcd((u - v)/2, v)$. If both are odd and $u < v$, then $\gcd(u, v) = \gcd((v - u)/2, u)$. These are combinations of one step of the simple Euclidean algorithm, which uses subtraction at each step, and an application of step 3 above. The division by 2 results in an integer because the difference of two odd numbers is even.^[3]
- Repeat steps 2–4 until $u = v$, or (one more step) until $u = 0$. In either case, the GCD is $2^k v$, where k is the number of common factors of 2 found in step 2.

The algorithm requires $O(n^2)$ ^[4] worst-case time, where n is the number of bits in the larger of the two numbers. Although each step reduces at least one of the operands by at least a factor of 2, the subtract and shift operations take linear time for very large integers (although they're still quite fast in practice, requiring about one operation per word of the representation).

An extended binary GCD, analogous to the **extended Euclidean algorithm**, is given by Knuth along with pointers to other versions.^[5]

Implementation [edit]

Recursive version in C [edit]

Following is a **recursive** implementation of the algorithm in C. The implementation is similar to the description of

the algorithm given above. It use two arguments u and v . All but one of the recursive calls are [tail recursive](#).

```
unsigned int gcd(unsigned int u, unsigned int v)
{
    // simple cases (termination)
    if (u == v)
        return u;

    if (u == 0)
        return v;

    if (v == 0)
        return u;

    // look for factors of 2
    if (~u & 1) // u is even
    {
        if (v & 1) // v is odd
            return gcd(u >> 1, v);
        else // both u and v are even
            return gcd(u >> 1, v >> 1) << 1;
    }

    if (~v & 1) // u is odd, v is even
        return gcd(u, v >> 1);

    // reduce larger argument
    if (u > v)
        return gcd((u - v) >> 1, v);

    return gcd((v - u) >> 1, u);
}
```

Iterative version in C [\[edit\]](#)

Following is an implementation of the algorithm in C, taking two (non-negative) integer arguments u and v . It first removes all common factors of 2 using identity 2, then computes the GCD of the remaining numbers using identities 3 and 4, and combines these to form the final answer.

```
unsigned int gcd(unsigned int u, unsigned int v)
{
    int shift;

    /* GCD(0,v) == v; GCD(u,0) == u, GCD(0,0) == 0 */
    if (u == 0) return v;
    if (v == 0) return u;

    /* Let shift := lg K, where K is the greatest power of 2
       dividing both u and v. */
    for (shift = 0; ((u | v) & 1) == 0; ++shift) {
        u >>= 1;
        v >>= 1;
    }

    while ((u & 1) == 0)
        u >>= 1;

    /* From here on, u is always odd. */
    do {
        /* remove all factors of 2 in v -- they are not common */
        /* note: v is not zero, so while will terminate */
        while ((v & 1) == 0) /* Loop X */
            v >>= 1;

        /* Now u and v are both odd. Swap if necessary so u <= v,
           then set v = v - u (which is even). For bignums, the
           swapping is just pointer movement, and the subtraction
           can be done in-place. */
        if (u > v) {
```

```

        unsigned int t = v; v = u; u = t;} // Swap u and v.
        v = v - u; // Here v >= u.
    } while (v != 0);

    /* restore common factors of 2 */
    return u << shift;
}

```

Efficiency [\[edit\]](#)

Akhavi and Vallée proved that, in theory, binary GCD can be about 60% more efficient (in terms of the number of bit operations) on average than the Euclidean algorithm.^{[6][7][8]} However, although this algorithm modestly outperforms the traditional Euclidean algorithm in real implementations (see next paragraph), its **asymptotic performance** is the same, and binary GCD is considerably more complex to code given the widespread availability of a division instruction in all modern microprocessors. (Note however that the division instruction may take a significant number of cycles to execute, relative to the other machine instructions.^{[9][10]})

Real computers operate on more than one bit at a time, and even assembly language implementations of binary GCD have to compete against carefully designed hardware circuits for integer division. Overall, [Knuth \(1998\)](#) reports a 20% gain over Euclidean GCD, on a version of [MIX](#) (Knuth's abstract model of a machine architecture) extended with binary shift and test operations.

For [arbitrary-precision arithmetic](#), neither the Euclidean algorithm nor the binary GCD algorithm are fastest, as they both take time that is a [quadratic function](#) of the number of input digits. Instead, recursive methods that combine ideas from the binary GCD algorithm with the [Schönhage–Strassen algorithm](#) for fast integer multiplication can find GCDs in near-linear time.^[11]

Historical description [\[edit\]](#)

An algorithm for computing the GCD of two numbers was described in the ancient Chinese mathematics book *The Nine Chapters on the Mathematical Art*. The original algorithm was used to reduce a fraction. The description reads:

"If possible halve it; otherwise, take the denominator and the numerator, subtract the lesser from the greater, and do that alternately to make them the same. Reduce by the same number."


This description looks like a normal Euclidean algorithm, but there is ambiguity in the phrase "if possible halve it".^[*citation needed*] The traditional interpretation is that this only applies when 'both' numbers are even, implying the algorithm is just slightly inferior Euclidean algorithm (for using subtraction instead of division). But the phrase may mean dividing by 2 should 'either' of the numbers become even, in which case it is the binary GCD algorithm.








See also [\[edit\]](#)

- [Euclidean algorithm](#)
- [Extended Euclidean algorithm](#)
- [Least common multiple](#)



References [\[edit\]](#)

- ↑ Stein, J. (1967), "Computational problems associated with Racah algebra", *Journal of Computational Physics* **1** (3): 397–405, doi:[10.1016/0021-9991\(67\)90047-2](#) , ISSN 0021-9991
- ↑ Knuth, Donald (1998), *Seminumerical Algorithms*, *The Art of Computer Programming* **2** (3rd ed.), Addison-Wesley, ISBN 0-201-89684-2
- ↑ In fact, the algorithm might be improved by the observation that if both *u* and *v* are odd, then exactly one of *u* + *v* or *u* − *v* must be divisible by four. Specifically, assuming *u* ≥ *v*, if ((*u* **xor** *v*) **and** 2) = 2, then gcd(*u*, *v*) = gcd((*u* + *v*)/4, *v*), and otherwise gcd(*u*, *v*) = gcd((*u* − *v*)/4, *v*).
- ↑ <http://gmpilib.org/manual/Binary-GCD.html>
- ↑ Knuth 1998, p. 646, answer to exercise 39 of section 4.5.2
- ↑ Akhavi, Ali; Vallée, Brigitte (2000), "AverageBit-Complexity of Euclidean Algorithms" , *Proceedings ICALP'00, Lecture Notes Computer Science 1853*: 373–387, CiteSeerX: [10.1.1.42.7616](#)
- ↑ Brent, Richard P. (2000), "Twenty years' analysis of the Binary Euclidean Algorithm" , *Millenial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in honour of Professor Sir Antony Hoare* (Palgrave, NY): 41–53 proceedings edited by J. Davies, A. W. Roscoe and J. Woodcock.
- ↑ Notes on Programming  by Alexander Stepanov
- ↑ Jon Stokes (2007). *Inside the Machine: An Illustrated Introduction to Microprocessors and Computer*

-
- Architecture* . No Starch Press. p. 117. ISBN 978-1-59327-104-6.
10.  Robert Reese, J. W. Bruce, Bryan A. Jones (2009). *Microcontrollers: From Assembly Language to C Using the PIC24 Family* . Cengage Learning. p. 217. ISBN 1-58450-633-4.
11.  Stehlé, Damien; Zimmermann, Paul (2004), "A binary recursive gcd algorithm", *Algorithmic number theory*  (PDF), Lecture Notes in Comput. Sci. **3076**, Springer, Berlin, pp. 411–425, doi:10.1007/978-3-540-24847-7_31 , MR 2138011 .

Further reading [\[edit\]](#)

- 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. Problem 31-1, pg.902.

External links [\[edit\]](#)

- NIST Dictionary of Algorithms and Data Structures: binary GCD algorithm
- Cut-the-Knot: Binary Euclid's Algorithm at cut-the-knot
- *Analysis of the Binary Euclidean Algorithm* (1976), a paper by Richard P. Brent, including a variant using left shifts
- "Dynamics of the Binary Euclidean Algorithm: Functional Analysis and Operators" (1998), a paper by Brigitte Vallée.

v · t · e	Number-theoretic algorithms
Primality tests	AKS test · APR test · Baillie–PSW · ECPP test · Elliptic curve · Pocklington · Fermat · Lucas · <i>LUCAS–LEHMER</i> · <i>LUCAS–LEHMER–RIESEL</i> · <i>PROTH'S THEOREM</i> · <i>PÉPIN'S</i> · 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) · <i>Special number field sieve (SNFS)</i> · 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
<i>Italics indicate that algorithm is for numbers of special forms · Smallcaps indicate a deterministic algorithm</i>	

Categories: Number theoretic algorithms