Last update: October 15, 2024 Translated From: e-maxx.ru

Euclidean algorithm for computing the greatest common divisor

Given two non-negative integers a and b, we have to find their **GCD** (greatest common divisor), i.e. the largest number which is a divisor of both a and b. It's commonly denoted by $\gcd(a, b)$. Mathematically it is defined as:

$$\gcd(a, b) = \max\{k > 0 : (k \mid a) \text{ and } (k \mid b)\}$$

(here the symbol "$\mid$" denotes divisibility, i.e. "$k \mid a$" means "k divides a")

When one of the numbers is zero, while the other is non-zero, their greatest common divisor, by definition, is the second number. When both numbers are zero, their greatest common divisor is undefined (it can be any arbitrarily large number), but it is convenient to define it as zero as well to preserve the associativity of $\gcd$. Which gives us a simple rule: if one of the numbers is zero, the greatest common divisor is the other number.

The Euclidean algorithm, discussed below, allows to find the greatest common divisor of two numbers a and b in $O(\log \min(a, b))$. Since the function is **associative**, to find the GCD of **more than two numbers**, we can do $\gcd(a, b, c) = \gcd(a, \gcd(b, c))$ and so forth.

The algorithm was first described in Euclid's "Elements" (circa 300 BC), but it is possible that the algorithm has even earlier origins.

Algorithm

Originally, the Euclidean algorithm was formulated as follows: subtract the smaller number from the larger one until one of the numbers is zero. Indeed, if g divides a and b, it also divides $a-b$. On the other hand, if g divides $a-b$ and b, then it also divides $a = b + (a-b)$, which means that the sets of the common divisors of $\{a, b\}$ and $\{b, a-b\}$ coincide.

Note that a remains the larger number until b is subtracted from it at least $\left\lfloor\frac{a}{b}\right\rfloor$ times. Therefore, to speed things up, $a-b$ is substituted with $a-\left\lfloor\frac{a}{b}\right\rfloor b = a \bmod b$. Then the algorithm is formulated in an extremely simple way:

$$\gcd(a, b) = \begin{cases}a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise.}\end{cases}$$

Implementation

```
int gcd (int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd (b, a % b);
}
```

Using the ternary operator in C++, we can write it as a one-liner.

```
int gcd (int a, int b) {
    return b ? gcd (b, a % b) : a;
}
```

And finally, here is a non-recursive implementation:

```
int gcd (int a, int b) {
    while (b) {
        a %= b;
        swap(a, b);
    }
    return a;
}
```

Note that since C++17, `gcd` is implemented as a standard function in C++.

Time Complexity

The running time of the algorithm is estimated by Lamé's theorem, which establishes a surprising connection between the Euclidean algorithm and the Fibonacci sequence:

If $a > b \geq 1$ and $b < F_n$ for some n, the Euclidean algorithm performs at most $n-2$ recursive calls.

Moreover, it is possible to show that the upper bound of this theorem is optimal. When $a = F_n$ and $b = F_{n-1}$, $gcd(a, b)$ will perform exactly $n-2$ recursive calls. In other words, consecutive Fibonacci numbers are the worst case input for Euclid's algorithm.

Given that Fibonacci numbers grow exponentially, we get that the Euclidean algorithm works in $O(\log \min(a, b))$.

Another way to estimate the complexity is to notice that $a \bmod b$ for the case $a \geq b$ is at least 2 times smaller than a, so the larger number is reduced at least in half on each iteration of the algorithm. Applying this reasoning to the case when we compute the GCD of the set of numbers $a_1, \dots, a_n \leq C$, this also allows us to estimate the total runtime as $O(n + \log C)$, rather than $O(n \log C)$, since every non-trivial iteration of the algorithm reduces the current GCD candidate by at least a factor of 2.

Least common multiple

Calculating the least common multiple (commonly denoted **LCM**) can be reduced to calculating the GCD with the following simple formula:

$$\operatorname{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)}$$

Thus, LCM can be calculated using the Euclidean algorithm with the same time complexity:

A possible implementation, that cleverly avoids integer overflows by first dividing a with the GCD, is given here:

```
int lcm (int a, int b) {
    return a / gcd(a, b) * b;
}
```

Binary GCD

The Binary GCD algorithm is an optimization to the normal Euclidean algorithm.

The slow part of the normal algorithm are the modulo operations. Modulo operations, although we see them as $O(1)$, are a lot slower than simpler operations like addition, subtraction or bitwise operations. So it would be better to avoid those.

It turns out, that you can design a fast GCD algorithm that avoids modulo operations. It's based on a few properties:

- If both numbers are even, then we can factor out a two of both and compute the GCD of the remaining numbers: $\gcd(2a, 2b) = 2 \gcd(a, b)$.
- If one of the numbers is even and the other one is odd, then we can remove the factor 2 from the even one: $\gcd(2a, b) = \gcd(a, b)$ if b is odd.
- If both numbers are odd, then subtracting one number of the other one will not change the GCD: $\gcd(a, b) = \gcd(b, a - b)$

Using only these properties, and some fast bitwise functions from GCC, we can implement a fast version:

```
int gcd(int a, int b) {
    if (!a || !b)
        return a | b;
    unsigned shift = __builtin_ctz(a | b);
    a >>= __builtin_ctz(a);
    do {
        b >>= __builtin_ctz(b);
        if (a > b)
            swap(a, b);
        b -= a;
    } while (b);
    return a << shift;
}
```

Notice, that such an optimization is usually not necessary, and most programming languages already have a GCD function in their standard libraries. E.g. C++17 has such a function `std::gcd` in the `numeric` header.

Practice Problems

- CSAcademy - Greatest Common Divisor
- Codeforces 1916B - Two Divisors

Contributors:

Nalin Bhardwaj (42.64%) jakobkogler (38.76%) adamant-pwn (10.85%) tcNickolas (3.88%) mdeapedro (1.55%) AniketR10 (0.78%) itssachinkr (0.78%) jylin34 (0.78%)