Daniel Lemire's blog

Daniel Lemire is a computer science professor at the University of Quebec (TELUQ) in Montreal. His research is focused on software performance and data engineering. He is a techno-optimist.

Fastest way to compute the greatest common divisor

Given two positive integers *x* and *y*, the greatest common divisor (GCD) *z* is the largest number that divides both *x* and *y*. For example, given 64 and 32, the greatest common divisor is 32.

There is a fast technique to compute the GCD called the binary GCD algorithm or Stein's algorithm. According to Wikipedia, it is 60% faster than more common ways to compute the GCD.

I have honestly never written a program where computing the GCD was the bottleneck. However, Pigeon wrote a blog post where the binary GCD fared very poorly compared to a simple implementation of Euler's algorithm with remainders:

```
unsigned gcd_recursive(unsigned a, unsigned b)
{
    if (b)
        return gcd_recursive(b, a % b);
    else
        return a;
}
```

Though Pigeon is a great hacker, I wanted to verify for myself. It seems important to know whether an algorithm that has its own wikipedia page is worth it. Unfortunately, the code on Wikipedia's page implementing the binary GCD algorithm is either inefficient or slightly wrong. So I wrote my own version using a GCC intrinsic function (`__builtin_ctz`) to find the *number of trailing zeros*:

```
unsigned int gcd(unsigned int u, unsigned int v)
{
    int shift;
    if (u == 0) return v;
```

```
    if (v == 0) return u;
    shift = __builtin_ctz(u | v);
    u >>= __builtin_ctz(u);
    do {
        v >>= __builtin_ctz(v);
        if (u > v) {
            unsigned int t = v;
            v = u;
            u = t;
        }
        v = v - u;
    } while (v != 0);
    return u << shift;
}
```

(Update: this code was improved due to a remark by Ralph Corderoy.)

My result? Using integers in [0,2000), the simple version Pigeon proposed does 25 millions GCDs per second, whereas my binary GCD does 39 millions GCDs per second, a difference of 55% on an Intel core i7 desktop. Why do my results disagree with Pigeon? His version of the binary GCD did not make use of the intrinsic `__builtin_ctz` and used an equivalent loop instead. When I implemented something similarly inefficient, I also got a slower result (17 millions GCDs per second) which corroborates Pigeon's finding.

My benchmarking code is available.

On a 64-bit machine, you probably can adapt this technique using the `__builtin_ctzll` intrinsic.

Update: You can read more about sophisticated GCD algorithms in the gmplib manual.

Conclusion: The Binary GCD is indeed a faster way to compute the GCD for 32-bit integers, but only if you use the right instructions (e.g., `__builtin_ctz`). And someone ought to update the corresponding Wikipedia page.

PUBLISHED BY

Daniel Lemire

A computer science professor at the University of Quebec (TELUQ). View all posts by Daniel Lemire →

29 thoughts on “Fastest way to compute the greatest common divisor”

Steven Pigeon

December 26, 2013 at 6:53 pm

What I don’t get is that you have a speedup only on numbers of the form mÃ—2^k and nÃ—2^j, and the speed-up is proportional to min(j,k).

How do you explain doubling the speed if asymptotically few pairs of numbers are of that form?

lecteur habituel

December 26, 2013 at 5:45 pm

euclyd, not euler.

Thanks for the post!

Maths Brane

December 12, 2018 at 9:18 pm

Yea, I heart Euler, but this is Euclid, all day.

Leonid Boytsov

December 26, 2013 at 6:21 pm

Another excellent example of shaving off constants!

Steven Pigeon

December 27, 2013 at 9:27 am

They’re not quadratic, they’re O(lg min(a,b)).

see:

http://en.wikipedia.org/wiki/Euclidean_algorithm#Algorithmic_efficiency

Per Persson

December 27, 2013 at 12:39 pm

“And someone ought to update the corresponding Wikipedia page.”

Why don’t you do it yourself?

Per Persson

December 27, 2013 at 1:32 pm

By the way, the numbers you used for testing are relatively small. More complicated algorithms are often slower for small numbers and don’t show their efficiency until the numbers are bigger. Without using anything bigger than uint32 you could test numbers of size ~1’000’000’000.

Mike

December 27, 2013 at 9:07 am

If you care about asymptotics, then both of these are quadratic. For a subquadratic algorithm, you need something like a half-gcd based algorithm.

Daniel Lemire ▲

December 27, 2013 at 9:51 am

@Pigeon

It is not necessary for the numbers to be divisible by two for them to benefit from the binary GCD.

Take 3 and 5. After the first pass in the loop you get 3 and 2. The 2 gets back to 1 due to the ctz shift.

The nice thing with the binary GCD is that it does not use any expensive operation (ctz is quite cheap on recent Intel processors) whereas the basic GCD relies on integer division.

Mike

December 27, 2013 at 10:04 am

They are quadratic when considering operations with integers that are larger than an unsigned int/long.

For example, see: https://gmplib.org/manual/Greatest-Common-Divisor-Algorithms.html and https://gmplib.org/manual/Binary-GCD.html

Ralph Corderoy

December 27, 2013 at 10:53 am

Hi Daniel, I can trim another 12% off your gcd() above by removing the two redundant shifts by “shift” of “u” and “v” that occur before the loop.

Daniel Lemire

December 27, 2013 at 11:10 am

@Ralph

Well done. I have updated my blog post and credited you for the gains.

KWillets

December 27, 2013 at 12:46 pm

I wonder if you could save a cmpl by reusing the u > v comparison for the loop break as well. That is:

if(u == v)
break;
else if (u > v)
…

This will shorten the last iteration and probably speed up the speculative execution.

Daniel Lemire

December 27, 2013 at 1:29 pm

@KWillets

With clang, your version is faster. With GCC, the version in the blog post is faster. The difference is within 10%.

If I played with compiler flags, there might be other differences as well.

In any case, your version is on github if you want to benchmark it.

Daniel Lemire

December 27, 2013 at 1:46 pm

@Persson

I have added a test in my code with large numbers but it makes no difference. Of course, these are word-size integers… results would differ with big integers.

Ralph Corderoy

December 28, 2013 at 5:57 am

Hi again Daniel, I can save a further 7.5% on my earlier suggestion by altering the loop to

```
do {
unsigned m;
v >>= __builtin_ctz(v);
m = (v ^ u) & -(v < u);
u ^= m;
v ^= m;
v -= u;
} while (v);
```

Steven Pigeon

December 28, 2013 at 12:37 pm

I have re-run tests with a version using the built-ins. The speed-ups are there indeed: 40% on larger numbers.

http://hbfs.wordpress.com/2013/12/10/the-speed-of-gcd/

(and @Mike I think the state of the art for fast division is O(n^log_2(3)), which is still more than linear, but subquadratic.)

KWillets

December 28, 2013 at 11:38 am

For my tweak the assembler output from gcc has the comparison, then a branch to the top of the loop, then the same comparison :(. The second comparison isn't reachable by any other path either.

Maybe some syntactic shuffling would trigger the optimization; I may give it a few tries later.

KWillets

December 28, 2013 at 8:22 pm

This is faster on my version of gcc:

```
{
int shift, uz, vz;
uz = __builtin_ctz(u);
if ( u == 0) return v;

vz = __builtin_ctz(v);
if ( v == 0) return u;

shift = uz > vz ? vz : uz;

u >>= uz;

do {
v >>= vz;

if (u > v) {
```

```
unsigned int t = v;
v = u;
u = t;
}

v = v – u;
vz = __builtin_ctz(v);
} while( v != 0 );

return u << shift;
}
```

Results:

gcd between numbers in [1 and 2000]
26.4901 17.6991 32.7869 25.974 24.3902 31.746 36.6972

I was actually trying to get it to utilize the fact that ctz sets the == 0 flag when its argument is 0, so a following test against 0 should not need an extra instruction. However the compiler didn't notice. Instead it set up some interesting instruction interleaving so that the v != 0 test is actually u == v before the subtraction; I believe this is to enable ILP.

Also, using an inline xchg instruction for the swap doubles the speed:

gcd between numbers in [1 and 2000]
26.1438 16.3934 33.6134 25.974 25.4777 30.5344 72.7273
gcd between numbers in [1000000001 and 1000002000]
26.1438 16 33.8983 25.974 25.3165 29.6296 72.7273

Daniel Lemire ♟

December 29, 2013 at 1:09 pm

@KWillets

Thanks. I have added your code to the benchmark.

Do you have the code for the version with the xchg instruction?

Daniel Lemire

December 29, 2013 at 1:09 pm

@Ralph

I added your version to the benchmark.

KWillets

December 29, 2013 at 1:23 pm

Here's the asm for the swap; I just replaced the part inside the brackets with xswap(u,v):

#define xswap(a,b) __asm__ (\
"xchg %0, %1\n"\
: : "r"(a), "r" (b));

Unfortunately I don't understand if this is correctly defined (I copied it from some poorly-documented examples), but the assembler output looks good.

Daniel Lemire

December 29, 2013 at 2:46 pm

@KWillets

I have checked into github a version with your inline assembly (slightly tweaked to be more standard). It is not faster.

When I ran your code "as is" I got failed tests.

https://github.com/lemire/Code-used-on-Daniel-Lemire-s-blog/blob/master/2013/12/26/gcd.cpp

---

KWillets

December 30, 2013 at 12:54 pm

Looking at Steven's asm listings, I realized that my compiler was significantly behind, so I downloaded 3G of Apple "updates" last night. These results are now from clang-

500.2.79.

I started playing around with various ways of getting abs(v-u) (especially when unsigned) and also realized that bsfl(x) == bsfl(-x), so this works for the inner loop on gcdwikipedia5fast:

```
do {
v >>= vz;
unsigned int diff =v;
diff -= u;
vz = __builtin_ctz(diff);
if( diff == 0 ) break;
if ( v < u ) {
u = v;
v = 0 – diff;
} else
v = diff;

} while( 1 );
```

If diff is signed 32-bit it's slightly faster, abs(diff) can be used, and the v < u test can be switched to diff < 0 for a slight gain. But it becomes a 31-bit algorithm. I haven't tried signed 64-bit yet.

Using bsfl(diff) instead of v seems to speed it up significantly; it's probably ILP again since it doesn't have to wait for v to finalize.

KWillets

December 30, 2013 at 1:02 pm

Hold on, I just tried signed 64-bit and got a huge boost:

```
do {
v >>= vz;
long long int diff = v ;
diff -= u;
vz = __builtin_ctz(diff);
if( diff == 0 ) break;
if ( diff < 0 )
```

```
u = v;
v = abs(diff);

} while( 1 );
```

Daniel Lemire

December 30, 2013 at 1:26 pm

@KWillets

I added these two alternatives to the benchmark.

I find that results vary a lot depending on the compiler and processor. It is hard to identify a clear winner… except that they are all faster than the Euclidean algorithm with remainder.

KWillets

December 30, 2013 at 2:37 pm

I checked the new revision and the 64-bit version (7) should use abs() and a few other edits.

Should I be submitting edits to github?

Taeseung Lee

December 30, 2016 at 3:19 pm

Thanks for the post!

detailyang

April 29, 2019 at 1:54 pm

It’s cool and it’ faster 3x than mod in my golang implement