Hyperbolic pairing function

--Steve Witham 2020-03-30.

Paper v0.4. Function v0.
The previous version, the first sent to others, had no version number.

Changes since then:

- Explicit about which "hyperbolas," in the intro.
- Consistently $f(x, y) = z$, $a(n) = c$.

Not fixed:

- Detail how " a^{-1} " $(z) = n$ but $a(n) \neq z$.

A pairing function is an $f(x, y)$ that takes two numbers (in our case any two positive integers) and somehow encodes them uniquely into a number z. Then an inverse function $f^{-1}(z)$ gives back the pair (x, y).

The granddad of pairing functions, Georg Cantor's, indexes pairs by scanning $y = n - x$ diagonals in an x, y grid. Here we define a pairing that scans $y = n/x$ hyperbolas that pass through integer points, for which the sequence $a(n)$ in [https://oeis.org/A006218 (https://oeis.org/A006218)](https://oeis.org/A006218) is helpful. This definition (one of several there)...

$$a(n) = \sum_{k=1..n} d(k),$$
$$\text{where } d(k) = \text{number of divisors of k},$$

means that the half-open interval $[a(n-1), a(n))$ has just enough room for those pairs whose product is n. If we but assign the pairs locations within the interval, a pairing function is defined.

Contents

Digression about harmonic numbers

Encoding $f(x, y)$

To encode a pair (x, y), first let

$$n = xy.$$

Arrange the prime-power factors of n in the usual way

$$n = \prod_j p_j^{t_j}$$

$$\text{where } p_k < p_j \text{ whenever } k < j.$$

x and y are products of different powers of the same p_j's:

$$x = \prod_j p_j^{r_j}$$

$$y = \prod_j p_j^{s_j}$$

Encode x's "r digits in base $(t + 1)$," *viz*, and bada-boom,

$$z = f(x, y) = a(n - 1) + \sum_j r_j \prod_{k<j} (t_k + 1).$$

(I believe this is easier than, e.g., enumerating all the possible x's and sorting them by x. And n.b., that would give a different ordering.)

Decoding

Given z, the encoded number, find n, the greatest number such that $a(n - 1) <= z$. Factor n arranging the primes in increasing order, then decode the "digits" of x:

$$\text{(for all the j's) } r_j = \lfloor \frac{z - a(n-1)}{\prod_{k<j} t_k + 1} \rfloor \mod t_j + 1$$

Finally, $x = \prod_j p_j^{r_j}$, and $y = n/x$.

Cost in bits

The encoded value of a pair is in a range

$$a(xy - 1) <= f(x, y) < a(xy).$$

There are some whole number n, m pairs for which $a(n) = 2^m$. In such cases an m-bit number can encode just the pairs from (1, 1) through those where $xy = n$. So it seems fair in general to say $f(x, y)$ "costs" $\log_2 a(xy)$ a.k.a. $\lg a(xy)$ bits.

We might expect the cost of $f(x, y)$ to be $O(\lg x + \lg y)$ bits, plus some overhead. What's the overhead? Skipping ahead some (see "calculating... approximately" below),

$$\lg f(x, y) = O(\lg(xy(\ln xy + 2\ \text{euler\_gamma} - 1)))$$

which indeed is

$$O(\lg x + \lg y + \lg(\ln xy + 2\ \text{euler\_gamma} - 1))$$

(Mumble about how the $\lg \ln xy$ part is the cost of choosing how many of n's bits belong to x vs. y, amortized across n's with different numbers of divisors.)

This "hyperbolic pairing" packs the number line without gaps, and assigns (x, y) pairs in xy order, which is to say in $(\lg x + \lg y)$ order. So I believe the "cost function" above, with its slightly mysterious overhead, is optimal for pairings that aim for that "$\lg x + \lg y$" property.

(Add some small and big examples.)

Calculating $c = a(n)$

Exactly

The formula I'm using for $a(n)$ takes $O(\sqrt{n})$ time:

$$a(n) = \left(\sum_{k=1}^{\lfloor \sqrt{n} \rfloor} \lfloor \frac{n}{k} \rfloor \right) - \lfloor \sqrt{n} \rfloor^2$$

Belatedly noticed that Charles R Greathouse IV gives this formula in PARI on the OEIS page. He also gives references to two $O(n^{1/3})$ methods. (And summarizes the proven bounds on the value of $a(n)$, below).

Approximately

The approximation everyone uses is:

$$a(n) \approx n(\log n + 2\ \text{euler\_gamma} - 1)$$

But see "digression about harmonic numbers" below.

Bounds on c

Having bounds on the function helps to invert the function.

From [https://oeis.org/A006218 (https://oeis.org/A006218)](https://oeis.org/A006218) :

> Let E(n) = a(n) - n(log n + 2 gamma - 1). Then Berkane-Bordellès-Ramaré show that
>
> - |E(n)| <= 0.961 sqrt(n),
> - |E(n)| <= 0.397 sqrt(n) for n > 5559, and
> - |E(n)| <= 0.764 n^(1/3) log n for x > 9994.
>
>   -Charles R Greathouse IV Oct 02 2017

It *seems* that $|E(n)| < 3n^{1/4}$. It certainly is for n <= 20000. Since $a(n)$ is monotonic, starting a search with those assumed bounds would quickly notice any exception. See also "approximating the inverse" in the next section.

Calculating the inverse $n = a^{-1}(c)$

This means search

I don't know a better answer than the Newtonish binary search I'm using, whose time is $O(\sqrt{n} \log \log n)$ or $O(n^{1/3} \log \log n)$. The square or cube root from the forward function is the worst contributor to this sorry situation. Having a good estimate and good bounds cuts the time (but only) by a constant factor. Also, it helps that $a(n)$ is strictly increasing (if fractal).

Approximating the inverse

One inverts the approximator. (See "approximately", above.) Although Newton's method would work, instead I cribbed this fixed-point method from Stack Overflow:

```
def inv_guess_a(c):
    if c < 2:
        return c

    n = c
    for k in range(10):
        n = c / (log(n) + 2 * euler_gamma - 1)
    return n
```

Bounds on n

Given c and the `inv_guess_a` function just above, my inverse search function gets itself rolling by setting high and low bounds on n, and a guess in the middle, like this:

```
delta_c = 3 * c**(1/4)
n_low_bound = inv_guess_a(c - delta_c)
n_guess = inv_guess_a(c)
n_high_bound = inv_guess_a(c + delta_c)
```

Fourth-root bounds mean that 3/4 of the result bits have already been found. But down in the low bits fractals loom, and estimates of the derivative get worse instead of better.

Digression about harmonic numbers

One of the definitions of $a(n)$ is

$$a(n) = \sum_{k=1}^{n} \lfloor n/k \rfloor$$

while that of the harmonic numbers is

$$H(n) = \sum_{k=1}^{n} 1/k.$$

And (this is mentioned on the OEIS page) using $H(n)$ gives a slightly better approximation to $a(n)$ (especially with the first few numbers) than the log-based approximation:

$$a(n) \approx n(H(n) + \text{euler\_gamma} - 1)$$

Meanwhile (this is exact)

$$H(n) = \text{digamma}(n + 1) + \text{euler\_gamma}$$

I guess the reason the log version is popular is that $H(n)$ only helps *approximate* $a(n)$, and the log approximates $H(n)$, so skip the middleman. Also, at least with the math libraries I have, the digamma takes fifteen times as long to run as the log does.