Hyperbolic pairing function

--Steve Witham 2020-03-31.

DRAFT v0.5. Function v0. Check the history at the bottom.

"In mathematics, a pairing function is a process to uniquely encode two natural numbers into a single natural number." [1] (https://en.wikipedia.org/wiki/Pairing_function)

Georg Cantor's pairing function indexes pairs by scanning $y = n - x$ diagonals in the x, y grid. Here we define a pairing $f(x, y)$ that collects (*positive*) integer points that $y = n/x$ hyperbolas pass through, for which the sequence $a(n)$ in OEIS A006218 (https://oeis.org/A006218) is very 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, since each (x, y) pair whose product is n can be identified with one of n's divisors, the range $a(n-1)..a(n)-1$ has just enough room to encode those pairs. If we assign each pair a location within the range, a pairing function is defined.

Contents

Encoding $f(x, y) => z$

The number of *ordered* pairs (x, y) whose product is n naturally matches the number divisors of n. For example,

$$6 = (1 \cdot 6) = (2 \cdot 3) = (3 \cdot 2) = (6 \cdot 1), \text{ and}$$
$$4 = (1 \cdot 4) = (2 \cdot 2) = (4 \cdot 1).$$

Knowing n, we can identify (x, y) by encoding its *first* number (that is, x) into an offset within the range of encoded numbers (z's) that belong to n. We must fix the order of the primes in the factoring of n in order to fix the definition of f. Let's use the usual order:

$$n = \prod_j p_j^{t_j}$$
$$\text{where } p_k < p_j \text{ whenever } k < j.$$

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

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

But each s_j is just $t_j - r_j$, and we ignore y for now. Given the r_j's and t_j's in our standard order, the following defines f. We work an example immediately below.

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

(Since the (x, y) points that go with n lie on a hyperbola, putting them in left-to right (i.e. x) order would seem natural but is more work, because we would still need to factor n, and then enumerate *all* its divisors and sort them. But note well that *this* method puts the pairs in a different order.)

Worked example

Let's encode the pair (1405, 18865).

$$\begin{aligned}
x &= 1405 \\
y &= 18865 \\
n &= xy = 25939375 \\
n &= 5^4 \cdot 7^3 \cdot 11^2 \\
x &= 5^3 \cdot 7^0 \cdot 11^1 \\
r_j\text{'s} &= 3, 0, 1 \\
t_j\text{'s} &= 4, 3, 2 \\
d(n) &= (4+1)(3+1)(2+1) = 60 \\
a(n-1) &= 446823997 \\
a(n) &= 446824057 \\
a(n) - a(n-1) &= 60
\end{aligned}$$

In this sum of products being added to $a(n-1)$,

$$z = a(n-1) + \sum_i r_j \prod_{k<i} (1 + t_k),$$

think of the r_j as the *digits*, and the (1+t_j) as the *base for each digit*. Just as decimal $300 = 3 \cdot 10 \cdot 10$, each digit is multiplied by the product of the bases *below* it. So,

$$\begin{aligned} a(n-1) &= 446823997 \\ t_j\text{'s} &= 4, 3, 2 \\ r_j\text{'s} &= 3, 0, 1 \\ z = f(x,y) &= 446823997 \ldots \\ &\quad + (r_1 = 3) \cdot (\text{empty product} = 1) \\ &\quad + (r_2 = 0) \cdot (1 + t_1 = 5) \\ &\quad + (r_3 = 1) \cdot (1 + t_1 = 5)(1 + t_2 = 4) \\ z &= 446823997 + 3 + 0 + 20 \\ &= 446824020 \end{aligned}$$

Notes

1. The last of the r_j's is the most significant (top) digit.
2. The base of the top digit $(1 + t_3 = 3)$, is not used in the calculation, but the corresponding prime is needed when factoring or recreating x.
3. The relation between x, which is 1405, and the offset, 23, is like this:

```
1405 -> factor -> powers -> digit coding -> 23
                ^   3,0,1         ^
                |                 |
                factors of 25939375
                |                 |
                v     3,0,1       v
1405 <- multiply <- powers <- decode digits <- 23
        p^r_j's
```

Exercise

There are 60 divisors of 25939375. Make a table of offsets from +0 to +59, each followed by the (x, y) pair that goes with that offset as we calculate it here. Already we have one finished!

```
+23: (1405, 18865)
```

But first: Guess where pair `(18865, 1405)` is in the list.
Question: Would you want to produce that table every time you encoded or decoded a pair?

Decoding $f^{-1}(z) => (x, y)$

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:

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

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 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": $\text{greatest } n : a(n-1) \le z$

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.

References

1. Wikipedia, Pairing function (https://en.wikipedia.org/wiki/Pairing_function)
2. OEIS, A006218 (https://oeis.org/A006218)

Changes in v0.5 since v0.4:

- Fixed the formula for $a(n)$ (!)
- In intro, ..."collects" (not scans) points that hyperbolas pass thru.
- Explains identification of pairs with x alone.
- Worked example of f(x, y), decode<->encode diagram, and an exercise.
- Using "reference-style" markdown links.
- Reference section
- Consistent scare-quoted "inverse" of a(n) is greatest n: a(n - 1) <= z.

Changes in v0.4 since the anonymous version:

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

The first version sent to others (and then only up thru "Encoding"), had no version number.