```
        x = y
        y = (x + n // x) // 2
    return x
```

Cantor pairing function is really one of the better ones out there considering its simple, fast and space efficient, but there is something even better published at Wolfram by Matthew Szudzik, here. The limitation of Cantor pairing function (relatively) is that the range of encoded results doesn't always stay within the limits of a `2N` bit integer if the inputs are two `N` bit integers. That is, if my inputs are two `16` bit integers ranging from `0 to 2^16 -1`, then there are `2^16 * (2^16 -1)` combinations of inputs possible, so by the obvious Pigeonhole Principle, we need an output of size at least `2^16 * (2^16 -1)`, which is equal to `2^32 - 2^16`, or in other words, a map of `32` bit numbers should be feasible ideally. This may not be of little practical importance in programming world.

Cantor pairing function:

```
(a + b) * (a + b + 1) / 2 + a; where a, b >= 0
```

> The mapping for two maximum most 16 bit integers (65535, 65535) will be 8589803520 which as you see cannot be fit into 32 bits.

Enter *Szudzik's function*:

```
a >= b ? a * a + a + b : a + b * b;  where a, b >= 0
```

> The mapping for (65535, 65535) will now be 4294967295 which as you see is a 32 bit (0 to 2^32 -1) integer. This is where this solution is ideal, it simply utilizes every single point in that space, so nothing can get more space efficient.

Now considering the fact that we typically deal with the signed implementations of numbers of various sizes in languages/frameworks, let's consider `signed 16` bit integers ranging from `-(2^15) to 2^15 -1` (later we'll see how to extend even the ouput to span over signed range). Since `a` and `b` have to be positive they range from `0 to 2^15 - 1`.

Cantor pairing function:

> The mapping for two maximum most 16 bit signed integers (32767, 32767) will be 2147418112 which is just short of maximum value for signed 32 bit integer.

Now *Szudzik's function*:

> (32767, 32767) => 1073741823, much smaller..

Let's account for negative integers. That's beyond the original question I know, but just elaborating to help future visitors.

Cantor pairing function:

```
A = a >= 0 ? 2 * a : -2 * a - 1;
B = b >= 0 ? 2 * b : -2 * b - 1;
(A + B) * (A + B + 1) / 2 + A;
```

> (-32768, -32768) => 8589803520 which is Int64. 64 bit output for 16 bit inputs may be so unpardonable!!

Szudzik's function:

```
A = a >= 0 ? 2 * a : -2 * a - 1;
B = b >= 0 ? 2 * b : -2 * b - 1;
A >= B ? A * A + A + B : A + B * B;
```

> (-32768, -32768) => 4294967295 which is 32 bit for unsigned range or 64 bit for signed range, but still better.

Now all this while the output has always been positive. In signed world, **it will be even more space saving if we could transfer half the output to negative axis**. You could do it like this for Szudzik's:

```
A = a >= 0 ? 2 * a : -2 * a - 1;
B = b >= 0 ? 2 * b : -2 * b - 1;
C = (A >= B ? A * A + A + B : A + B * B) / 2;
a < 0 && b < 0 || a >= 0 && b >= 0 ? C : -C - 1;

(-32768, 32767) => -2147483648

(32767, -32768) => -2147450880

(0, 0) => 0

(32767, 32767) => 2147418112

(-32768, -32768) => 2147483647
```

What I do: After applying a weight of `2` to the the inputs and going through the function, I then divide the ouput by two and take some of them to negative axis by multiplying by `-1`.

See the results, for any input in the range of a signed `16` bit number, the output lies within the limits of a signed `32` bit integer which is cool. I'm not sure how to go about the same way for Cantor pairing function but didn't try as much as its not as efficient. **Furthermore, more calculations involved in Cantor pairing function means its slower too**.

Here is a C# implementation.

```
public static long PerfectlyHashThem(int a, int b)
{
    var A = (ulong)(a >= 0 ? 2 * (long)a : -2 * (long)a - 1);
    var B = (ulong)(b >= 0 ? 2 * (long)b : -2 * (long)b - 1);
    var C = (long)((A >= B ? A * A + A + B : A + B * B) / 2);
    return a < 0 && b < 0 || a >= 0 && b >= 0 ? C : -C - 1;
}

public static int PerfectlyHashThem(short a, short b)
{
    var A = (uint)(a >= 0 ? 2 * a : -2 * a - 1);
    var B = (uint)(b >= 0 ? 2 * b : -2 * b - 1);
    var C = (int)((A >= B ? A * A + A + B : A + B * B) / 2);
    return a < 0 && b < 0 || a >= 0 && b >= 0 ? C : -C - 1;
}
```

Since the intermediate calculations can exceed limits of `2N` signed integer, I have used `4N` integer type (the last division by `2` brings back the result to `2N`).

The link I have provided on alternate solution nicely depicts a graph of the function utilizing every single point in space. *Its amazing to see that you could uniquely encode a pair of coordinates to a single number reversibly! Magic world of numbers!!*

▶ Is this even possible?

You are combining two integers. They both have the range -2,147,483,648 to 2,147,483,647 but you will only take the positives. That makes 2147483647^2 = 4,61169E+18 combinations. Since each combination has to be unique AND result in an integer, you'll need some kind of magical integer that can contain this amount of numbers.

Or is my logic flawed?
