Hypernumbers - hidden in plain sight

Conrad sen Kyne

What is a hypernumber?

In order to count in any base, we define a character set of digits which represent each possible value from ”zero” to the number right before ”ten”.

$base10$	$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$
$base2$	$0, 1$
$base7$	$0, 1, 2, 3, 4, 5, 6$
$base16$	$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$

I assume if you’re reading this, you already know how to count, and you may even know how to count in binary, octal (base 8), or hexadecimal. But what would we do if we wanted to use base 100? Or base 750? Would we have to make a bunch of unique characters and order them, then memorize them all? How would we program a computer to do this? Sure, there’s ASCII and unicode, but then how would humans reliably distinguish between an e with an umlaut and an e with a tilde at a glance, even if you memorized the character order? Would umlaut-ed characters be consecutive, or would a-like, b-like, c-like, etc characters be grouped consecutively?

To get started, let’s look at an interesting phenomenon: IP addresses. An IPv4 address contains 4 numbers between 0 and 255, separated by periods. IPv6 contains 8 numbers between 0 and 65535, represented in hexadecimal, and separated by colons. Even MAC addresses are hexadecimal values between 0 and 255 separated by a colon. They have a few common factors:

- The numbers are multiple digits long
- They are separated by some character
- They can be in any base

So what can we do with these? Well, we could try and construct a custom ”address” by emulating this structure. We do this by selecting three components for this address:

- A base for the separated numbers

- A maximum value for each number
- A unique non-numerical character to separate the numbers

Let's use 3, 6, and ":" respectively. In base 3, we count from 0 to 6 as follows:

$$0, 1, 2, 10, 11, 12, 20$$

If our address contains two numbers, we can select (with replacement) any two numbers from 0 to 20, giving us 49 total possible values, such as:

$$20:2 \quad 10:1 \qquad\qquad 12:12$$

If we wanted to generate this algorithmically, the output would probably look like this:

$$\begin{array}{cccccccc} 0:0 & 0:1 & 0:2 & 0:10 & 0:11 & 0:12 & 0:20 \\ 1:0 & 1:1 & 1:2 & ... \\ ... \\ 20:0 & 20:1 & 20:2 & ... \end{array}$$

In essence, we start with a 0 for the left number, then iterate from 0 to 20 on the right. Then we increase the left by 1 and repeat the process until we reach 20:20, the 49th number generated by this algorithm. If we had chosen base 7 for the individual numbers, we would have 6:6, and 66 is equal to 48 in base 7.

What we have done here is emulated the numbers from 0 to 48 in base 7 while only using the digits 0, 1, and 2, along with a unique character to distinguish between numbers. This structure is a hypernumber. The three components we used to build it are the base of the individual numbers (which we'll call *digits* from now on), the implicit base; the number which determines the maximum value of each digit, the hyperbase; and the separator or *separatrix* if you're feeling fancy.

By constructing hypernumbers, we can represent higher bases with fewer characters at the cost of more digits. Let's look at our inspiration again: IPv4 addresses are hypernumbers with hyperbase 256, implicit base 10, and separator ".". IPv6 is hyperbase 65536, implicit base 16, and separator ":". And MAC addresses are hyperbase 256, implicit base 16, and separator ":". Like with IPv6 addresses, hypernumbers can omit zeroes as long as the separators remain:

$$10, 0, 1, 0, 2 \rightarrow 10, , 1, , 2$$

They're already here

When we say numbers like 150200, we pronounce it "one hundred fifty thousand two hundred". We also often add a comma every 3 digits for readability, as 1,000,000,000 is easier to read than 1000000000. This also affects the way we name these numbers. We have unique groups for one, ten, and hundred, and thousand, but after that we only give a unique name to every number 3 digits larger (million, billion, trillion, etc). If we think about the comma (or sometimes period) separating every 3 digits as the separator of a hypernumber, we can derive that it is hyperbase 1000, implicit base 10, and separator "," or ".". That's right - this convention technically forms a hypernumber. Not only are they present in hardware and internet addresses, they're present in the language we use with large numbers!

This isn't some creative justification for the utility of hypernumbers, but a *demonstration* of this natural extension of the positional numeral system which is conducive to human understanding and interpretation of numbers. And it creeps in in other ways too. For example, how would you read this number?

$$1024$$

The proper name is "one thousand twenty-four", but it's also called "ten twenty-four", especially if the number represents a year. If we separated it according to this language, we would get 10:24, which is hyperbase 100, implicit base 10, and separator ":". Incidentally, that reminds me of the time, where the minutes can vary from 0 to 59, and the hours from 1 to 12 or 0 to 23. The implicit base is clearly 10.

These examples illustrate that this isn't a personal invention or even a discovery, really. Hypernumbers already exist and have existed for a while. And while certain numerical considerations exist (for example, we can calculate the number of unique IPv4 addresses, which happens to be 256^4) this concept isn't really discussed at all. I think this is because in mathematics, positional numeral systems are simply a representation of quantity. It's not as important that the system is deeply examined, just that it's used conventionally to guarantee people are on the same page. What is missing is a sociological discussion on how they came to be so ubiquitous, and how our conventions may be (and already are) augmented by hypernumbers.

Unresolved numbers

Let's say I told you that this number was in base 2:

$$130$$

How do we interpret it? 3 is not part of the digit set, but knowing the value of 3 in base 10, we can still compute its potential value:

$$130 \rightarrow 0 \cdot 1 + 3 \cdot 2 + 1 \cdot 4 = 0 + 6 + 4 = 10$$

We could also try translating 3 to base 2, giving us 11. But if we wrote it that way:

$$1110$$

Then the resultant value is $2 + 4 + 8 + 16 = 30$. We could even make it a hypernumber by adding a separator (making this into a hypernumber) giving us:

$$1, 11, 0$$

but that doesn't change the fact that 11 is still larger than the maximum value of 1. What we need to do is *resolve* this number to its canonical form.

Digit resolution is the process of carrying the value of a digit up, then resetting the digit down below the hyperbase. In essence, there is an invisible "final" character with the actual highest value. This value is always written as 10, but if the base were higher, we would use A (similar to how we use 3 even though we're in base 2). Mechanically and logically, we are counting from 0 to 1, 1 to 2, and so on until we count from 9 to 10. When we reach this point, it triggers us to carry up the value to resolve the number to its "true" form:

$$A_{10} \rightarrow 10$$

$$2_2 \rightarrow 10$$

$$5_5 \rightarrow 10$$

We don't *have* to resolve a number when it reaches this point, but we lose important properties of positional numeral systems: we lose uniqueness (eg $3_2 = 11_2$, and $5_2 = 21_2 = 101_2$) and we are forced to either expand the character set or treat the number as a hypernumber by giving it a separator. Hypernumbers can be convenient, but these problems are rarely solved with arbitrary complexity.

Sequences as baseless hypernumbers

What if we elected to use hypernumbers, but never resolved them? We could end up with arbitrary sequences like this:

$$3, 1, 9, 46, 2, 13$$

If we chose to never resolve any digit, regardless of how high it climbed, then we are *basically* saying that this sequence has an implicit base and a separator, but not a hyperbase. It's also common to represent sequences as polynomials, in much the same way as we represent numbers:

$$3x^0 + 1x^1 + 9x^2 + 46x^3 + 2x^4 + 13x^5$$

A sequence represented in this way is called a generating function, a tool which is invaluable to combinatorics. By leaving x (our hyperbase) as a variable, we can still interpret and even perform arithmetic on it while retaining its quantifying properties. We don't even have to call this a hypernumber, since it only has one base rather than two. After all, as we've seen with commas in large numbers, we can use separators even when we're not explicitly constructing hypernumbers, solely for the utility of readiblity.

Sequence arithmetic is nearly identical to ordinary arithmetic. Take, for example, the following expression:

$$1, 1 \cdot 1, 1$$

To solve this, we start by converting the sequences to polynomial form:

$$(1 + x) \cdot (1 + x)$$

Then we perform the FOIL method taught in algebra classes, multiplying each value of the left side by each of the right:

$$(x \cdot 1) + (1 \cdot x) + (x \cdot 1) + (x \cdot x) = 1 + x + x + x^2 = 1 + 2x + x^2$$

Converting back to sequence form, we have:

$$1, 2, 1$$

If we chose to add a hyperbase of 10 to this sequence, then the value is equivalent to 121 or 11^2. But we can interpret it in any hyperbase and it will be arithmetically consistent:

$$121_2 = 1001 = 9_{10} = (11_2)^2$$
$$121_3 = 121 = 16_{10} = (11_3)^2$$
$$121_4 = 121 = 25_{10} = (11_4)^2$$

If we instead multipled $3, 1 \cdot 3, 1$, the result would be $9, 6, 1$:

$$169_2 = 1101 = 25_{10} = (101_2)^2$$
$$169_3 = 1100 = 36_{10} = (20_3)^2$$
$$169_4 = 301 = 49_{10} = (13_4)^2$$

With larger sequences, we generalize the FOIL method as the sum of every digit of A times every digit of B, but the result is the same; $1, 1, 1 \cdot 1, 1, 1$ is correct in any base (the result happens to be $1, 2, 3, 2, 1$).

When assigning a base to a sequence, we can eliminate the separator after we have finished resolving it:

$$4, 1 \cdot 4, 1 = 16, 8, 1 = 6, 9, 1 = 196_{10}$$

Resolving negative digits

How do we interpret this sequence in base 10?

$$-2, -1, 1$$

Most easily, we can just evaluate the polynomial:

$$(1 \cdot -2) + (10 \cdot -1) + (100 \cdot 1) = -2 + -10 + 100 = 88$$

But we can also perform an uncarry, the reverse of a carry:

$$\begin{aligned} -2, -1, 1 = -2, -1 + 10, 0 &= -2, 9, 0 = -2, 9 \\ -2, 9 = -2 + 10, 8 &= 8, 8 \end{aligned}$$

There's a quirk I haven't mentioned yet: while sequences are read from left to right, interpreting it as a number requires reading it right to left. There's no secret reason for this, it's just our convention. If you know you're talking exclusively about digit sets, it's acceptable to reverse it so it also reads left to right. Do what's comfortable for you!

Uncarries are most evident when we perform subtraction. For the expression $11 - 9$, we compare the digits in matching places and difference them one at a time. When the top digit is smaller than the bottom digit, we subtract one from the digit on its left, add 10 to the current digit, *then* compute the difference. This is equivalent to the expression $(1 + 10) - 9$. But we can do this in reverse as well; we could subtract $1 - 9 = -8$, see the value is less than zero, and then decide to uncarry 10 into it, with the expression $(1 - 9) + 10$ (which works because addition is commutative, and subtraction is adding a negative).

Using sequences as a base

We've seen the impact of bases and hyperbases, but there is a mechanism we can exploit to count in a further unique way; instead of numbers like 10 or 16, we can use an entire sequence as a base. Let's reframe a typical base to this effect. Normally, with polynomial representation, each digit is multiplied by $1, b, b^2, b^3$, and so on. We can put this sequence into a function which we'll call f:

$$f(n) = b^n$$

Then our polynomial representation will look like this for 1024:

$$1024 = 4 \cdot f(0) + 2 \cdot f(1) + 0 \cdot f(2) + 1 \cdot f(3)$$

To turn this into a generating function, we use the powers of x:

$$1024 = 4 \cdot f(0) \cdot x^0 + 2 \cdot f(1) \cdot x^1 + 0 \cdot f(2) \cdot x^2 + 1 \cdot f(3) \cdot x^3$$

When x is not assigned a value, this is exactly a generating function. This representation is used not *so* arithmetic can be extended, but *because* it can be. By generalizing in this way, we open up a new world of opportunity. Before we continue, there are two important conditions that our function f must satisfy in order for this to work: the value of f(0) must always be 1; and the value of f(n) must always be greater than or equal to f(n-1).

Let's call our base sequence b:

$$b = 1, 2, 3, 4, 5, ...$$

So b(0) = 1, b(2) = 3, b(9) = 10, and so on. We then use subscript to denote the base as normal:

$$10_b$$

As we saw previously, interpreting this number is straightforward:

$$10_b = 0 \cdot f(0) + 1 \cdot f(1) = 0 \cdot 1 + 1 \cdot 2 = 2$$

Looks good so far. But how do we resolve these numbers while counting? The patterns aren't as obvious, but it's the same way we count with numerical bases:

- Increase the rightmost digit by 1
- Count the number of digits in the number, and call that n
- If the value of the sub-number from the 0th to the (n-1)th digit is equal to $f(n)$, change all of those digits to 0, and add a 1 as the new leftmost digit
- If the value is still less than f(n), repeat this process for the 0th to (n-2)th digits, checking if it is equal to f(n-1), then the 0th to (n-3)th checking f(n-2), and so on.

Let's try with a simple example with our original sequence b, starting with 0:

$$0 \rightarrow 1 \rightarrow 2$$

$1 \cdot f(0)$ is 1, which is less than f(1). But since $2 \cdot f(0) = 2 \cdot 1 = f(1)$, we resolve the number to 10.

$$\begin{aligned} 10 &\to 11 & &\to 100 \\ 100 &\to 101 & &\to 1000 \\ 1000 &\to 1001 & &\to 10000 \end{aligned}$$

Since $f(n) = f(n-1) + 1$, any number of the form 10...01 is resolved. Patterns like this are very common when using sequence bases, but usually more complex. Let's use the Fibonacci numbers now:

$$f = 1, 1, 2, 3, 5, 8, 13, ...$$

Let's count!

$$0 \to (1 \to 10)$$

$1 = f(1)$, so we resolve straight to 10. Whenever we are at f(n) but f(n+1) and f(n+2) are equal, then the (n+1)th digit will always be 0.

$$10 \to 11$$

$1 + 1 = f(2)$, so we again resolve to 100.

$$\begin{aligned} 100 &\to 101 & &\to 110 \to 1000 \\ 1001 &\to 1010 \\ 1011 &\to 1100 \to 10000 \end{aligned}$$

If you already know the Fibonacci numbers, the pattern is apparent; whenever you see two 1s next to each other, the number will always resolve upwards. This is because of how the sequence is defined:

$$\begin{aligned} f(0) &= f(1) = 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

This is called a recurrence relation. There are many sequences defined like this with similar counting patterns. The Lucas numbers, another well-known recurrence, is defined as:

$$\begin{aligned} f(0) &= 1 \\ f(1) &= 2 \\ f(n) &= 2 \cdot f(n-1) + f(n-2) \end{aligned}$$

Since the recurrence relation has the coefficients 2 and 1, we resolve the pair 21 like we resolve the pair 11 with the Fibonacci base:

$$1 \to (2 \to 10) \to 11 \to (12 \to 20) \to (21 \to 100)$$

Other recurrences will resolve their own coefficients' patterns:

$$\begin{array}{lr} f(n) = 3 \cdot f(n-1) + f(n-2) & 31 \to 100 \\ f(n) = 2 \cdot f(n-1) + 3 \cdot f(n-2) & 23 \to 100 \\ f(n) = 4 \cdot f(n-1) + 2 \cdot f(n-2) + f(n-3) & 421 \to 1000 \end{array}$$

With the third example, you can see the pattern quietly applies to longer recurrences. But there's something odd happening with the second one. The recurrence creates the sequence:

$$f = 1, 2, 7, 20, 61, ...$$

If we counted according to our original algorithm, we would have:

$$21 \to (22 \to 30) \to (31 \to 100)$$

When we reach 22, we have $2 = f(1)$ so we are <u>supposed</u> to resolve to 30, then resolve $1 + 2 \cdot 3 = f(2)$ to get 100. In theory, we would never actually see a 23 in this base.

This is where unresolved numbers come back into play. With a recurrence relation, it's much easier to recognize the pattern and count that way. We don't necessarily have all the values of f memorized (even when $f(n) = b^n$, except of course when $b = 10$) but that doesn't always restrict our counting. When we find these patterns, it makes sense to use them, and to delay resolution until the pattern appears. Any time we have a set of coefficients $\{a, b, c, ...\}$ which describe a recurrence relation, if $b > a$, or $c > b$, etc, this delayed resolution remains a possibility.

When we use a sequence as a base, it also has an implicit base, as you'll note that we've used an implicit base of 10 for digits. For this reason it is useful to distinguish numbers with sequence bases as hypernumbers (which don't need a separator like 1,000,000, but are still interpreted the same way). In fact, both the hyperbase and implicit base can be sequences. If both were the Fibonacci base, our number would look something like this:

$$10, 0, 10, 0$$

Where $10_f = 1_{10}$.

Near divisible forms and base interpretations

You might be familiar with the 9 divisibility test: Take a number, like 729, and add up its digits ($7 + 2 + 9 = 18$). Then ask whether that digit sum is divisible by 9. 18 is divisible by 9, and that means 729 is too. It also happens that if the digit sum is divisible by 3, so is 729. If this doesn't sound plausible, try experimenting with a few numbers in your calculator. You'll find that whenever the digit sum is NOT divisible by 9, neither is the number in question.

This doesn't work for every number. For example, 16 has a digit sum of 7, but it is not divisible by 7. But there *is* a divisibility test for 7 which makes use of multiple base interpretations of the same digit sequence.

The key is in understanding what a digit sum is. It's actually a base 1 interpretation of a (usually) unresolved number. With 729, we can represent it with $9 \cdot f(0) + 2 \cdot f(1) + 7 \cdot f(2)$, where $f = 1, 1, 1, 1, 1, ... = 1^n$. If we subtract the digit sum from the original number, that result is *also* divisible by 9. And maybe you're already seeing where I'm going with this.

Let's try interpreting a different number, 126, in base 2 and subtracting the result:

$$126_2 = 6 \cdot 1 + 2 \cdot 2 + 1 \cdot 4 = 6 + 4 + 4 = 14$$
$$126 - 14 = 112$$

112 is divisible by 2, 4, 7, 8, 14, 16, and 28...Oh? What's this? 112 is divisible by 8, and $10 - 2 = 8$. Mathematically, we say that 8 *divides* 112 with the following notation:

$$8|112$$

But 126 is not divisible by 8. Let's try again with 136:

$$136_2 = 6 \cdot 1 + 3 \cdot 2 + 1 \cdot 4 = 6 + 6 + 4 = 16$$
$$136 - 16 = 120$$

All three of 136, 16, and 120 are divisible by 8, or $10 - 2$. Coincidence? Well...let's try a base 5 interpretation of 125:

$$125_5 = 5 \cdot 1 + 2 \cdot 5 + 1 \cdot 25 = 5 + 10 + 25 = 40$$
$$125 - 40 = 85$$

In this case, all three numbers are divisible by 5, or $10 - 5$. We can even do this with 0. The power 0^0 is often defined as or argued to be 1, and we will use that definition here for a "base 0" interpretation:

$$126_0 = 6 \cdot 1 + 2 \cdot 0 + 1 \cdot 0 = 6 + 0 + 0 = 6126 - 6 = 120$$

The result is divisible by 10 this time, or $10 - 0$. With 120, we get 120, 0, and 120, all of which are divisible by $10 - 0$. So what's happening here?

As it turns out, if we have two bases b and t, then the difference between the two representations is always divisible by $b - t$. Consequently, this means that if the base t interpretation is divisible by $b - t$, then the base b interpretation is as well. We can prove this with modular arithmetic. For the expression $a\%b$, the solution is the remainder after dividing a by b:

$$\begin{aligned} 5\%2 &= 2r1 \\ 10\%3 &= 3r1 \\ 6\%4 &= 1r2 \end{aligned}$$

Modular arithmetic can perform algebraic manipulation of an equation. If $a + b + c = d + e + f$, then $(a + b + c)\%n = (d + e + f)\%n$. Even deeper, this distributes over addition, subtraction, and multiplication:

$$(a\%n) + (b\%n) + (c\%n) = (d\%n) + (e\%n) + (f\%n)$$

With modular arithmetic, we can prove the following statement:

$$(b - t)|N_t \iff (b - t)|N_b$$

Formally, this says "$b - t$ divides the base t interpretation of the digit set N *if and only if* $b - t$ also divides the base b interpretation." Remember that $b - t$ always divides the difference between the base b and base t interpretations of N:

$$(b - t)|(N_b - N_t) \quad \rightarrow (N_b\%(b - t)) - (N_t\%(b - t)) = 0$$

Given that $N_t$is divisible by $b - t$:

$$(N_b\%(b - t)) - 0 = 0 \quad \rightarrow (N_b\%(b - t)) = 0$$

Thus N_b is divisible by (b-t). This is proof that the difference between base interpretations is a near divisible form of the base b interpretation. So far it has been assumed that $(b - t)|(N_b - N_t)$, but before the proof is sound, we need something which proves this "given" fact. There is, in fact, an equality which can calculate the result of:

$$\frac{N_b - N_t}{b - t}$$

To showcase it, we will have to use summation notation. The summation operator uses the letter capital sigma to denote its operation. Here is a small example:

$$\sum_{x=0}^{10} f(x)$$

The expression $x = 0$ indicates the starting value of the variable x for this sum, while 10 indicates the maximum (inclusive) value of x. $f(x)$ is the inner function of the sum. To solve a sum, we iterate over all integer values of x between 0 and 10, compute the value of the inner function with that x, then add all the results together. The sample sum can be expanded:

$$f(0) + f(1) + f(2) + ... + f(9) + f(10)$$

An important convention with sums is that if the end value is lower than the start value, then the result of the sum is 0. This is called the empty sum.

$$\sum_{x=1}^{0} \int_{-x}^{x} e^{-k^2} dk = 0$$

This convention was decided similarly to how the factorial of 0 was set at 1. It just fits better, it's intuitive, and VERY practical.

The sums we're about to see may look intimidating because it involves a nested sum, a sum whose inner function itself contains a sum. But I want you to see it, in part to prove that it exists in the first place, and also to make summation notation more familiar. It will use the variable d to represent the digit set, so if $d = 6, 2, 1$ then $d(0) = 6$ and $d_b = 126_b$. We will also use the cardinality function $|d|$, which represents the number of elements in d.

Alright, no more stalling. The Near Divisible Form equality is:

$$\frac{\sum_{x=1}^{|d|} d(x) \cdot (b^x - t^x)}{b - t} = \sum_{x=1}^{|d|} d(x) \cdot \sum_{k=0}^{x-1} b^k t^{x-1-k}$$

The lefthand sum represents the difference between the base b and base t interpretations of each digit of d, all added up. To solve the righthand sum, we iterate from $x = 1$ to $|d|$, then during each iteration we iterate from $k = 0$ to $x - 1$. It can be very useful to create a table to organize this nested iteration:

x	d_x	k	$b^k t^{x-1-k}$	
1	3	0	$10^0 \cdot 3^{1-1-0}$	$= 1$
				$3 \cdot 1 = 3$
2	2	0	$10^0 \cdot 3^{2-1-0} = 1 \cdot 3$	$= 3$
		1	$10^1 \cdot 3^{2-1-1} = 10 \cdot 1$	$= 10$
				$2 \cdot (3 + 10) = 2 \cdot 13 = 26$
				$3 + 26 = 29$

Finally, Computing the NDF:

$$230 - (0 \cdot 1 + 3 \cdot 3 + 2 \cdot 9 = 27) = 230 - 27 = 203$$

And by calculator, $203/29 = 7$. It worked!

The NDF equality holds the same for hypernumbers, but only when the implicit base of both interpretations is the same. If the implicit bases differ, then the digit set being analyzed is fundamentally different, so subtracting one from the other will have arbitrary results that aren't NDFs.

Hyperhypernumbers

Yes, you read that right. While a hypernumber has 2 bases, numbers as digits, and a separator, a hyperhypernumber has an additional hyperhyperbase with hypernumbers as digits. The hyperseparator must also be distinct from the hypernumber's separator to avoid misinterpretation.

$$2, 11, 30 : 1, 21, 0$$

Remember, the implicit base, hyperbase, and hyperhyperbase can also be sequences! However, if the implicit base is a sequence, then its digits also have an implicit base, making it one layer more complex. The possibilities are endless. Converting the value of a hyperhypernumber to base 10 is quite obfuscated, but just follow these steps:

- convert the digits to the hyperbase and compress
- compute the hyperdigits in base 10 (or whichever base you feel most comfortable with)
- compute the value of the hyperhypernumber (perhaps with its polynomial representation)

$$hh = 13 \quad h = 3 \quad b = 2$$

$$\begin{aligned} 1,1,1:1,1,1 -> 111:111 &\qquad -> 13_{10}:13_{10} \\ 13 \cdot 1 + 13 \cdot 13 = 13 + 169 &\qquad = 182 \end{aligned}$$

So the value of $1,1,1:1,1,1$ is 182_{10}.

And yes...you can go further, and in fact indefinitely. If we avoid calling them hyperhyperhyperhypernumbers, and refer to their rank according to how many bases they have (or how many "hyper"s would appear in their name) then numbers are rank 0 hypernumbers, hypernumbers are rank 1, hyperhypernumbers are rank 2, and so on. I won't compute the base 10 value of this one, but this is a rank 3 hypernumber:

$$1,1:1,1:1,1/1,1:1,1:1,1$$

And yes, the NDF equality still applies. For a rank n hypernumber, as long as the rank 0 through rank (n-1) hyperbases are the same, then the difference is an NDF.

Conclusion

We've come far from the beginning of this article about hypernumbers. From the reverse-engineering of IP addresses; to the present existence of hypernumbers in society; to the interesting extensions of the base; to the surprising relation between interpretations of the same number. And there is still a lot we can ask:

- When, where, and how did colloquial usage of hypernumbers appear?
- What other instances are there which utilize hypernumbers?
- Where else would it be useful to incorporate hypernumbers?
- Are there colloquial instances of higher-rank hypernumbers? If not, is there any potential application of them in the present day?
- Is there a ranked version of NDF, where the rank 0 to (n-1) hyperbases may differ for a rank n hypernumber?

These questions are not only mathematical, they are sociological, because this article is not just describing a concept and pursuing proofs, but also aims to explain the social facts of hypernumbers' existence. With an interdisciplinary approach, we learn more about the *why* of positional numeral systems, which are definitively the most important and ubiquitous mathematical concept in human society.