Welcome to Quantum Notation, The Most Universal Counting Notation Created. This notation is created to represent smaller and bigger stages of the numbers 0 through Eternity, inside an array! The notation is created by Jesse Kohn.
Want to create subsubsubquantum numbers of Jesse's famous quantumnumbers Monero, Dero, etc.? Use this notation to help make it clear what values they are! Or, do you want to do the opposite and define as many Fictional 2 Googology, Fictional 3 Googology, and higher fictional numbers that equal to any number between Absolute Infinity and Eternity in higher realms? This notation also covers that!
You can mix and match any values in any realm, like 1 in one realm and ??? [32768] in another. The choice is up to you!

Structure

The notation is built to make it as easy as possible to understand and use.
First, we need to know our planes.

q(y;x) is our basic starting point. The y here is our quantum value for meta-quantum googology. In a different way, this is our bitsy value that lies inside of x. x is our normal number value. y and x are the bare minimum that must be present at once.

A value of q(2;0) means this is quantum two (2) in normal zero (0).
This is also known as Dero.

A value of q(10;2) means there is a quantum ten (10) in normal two (2).
This is also known as Dectwo.

In this equation, the right most value in the array is the base with the semicolons separators. You can use numbers from zero all the way to eternity (□) in any of the planes. Whenever a plane hits eternity (□), that plane resets to zero (0) and increments the next entry in the right by one (1). This means that q(□;0) will become q(0;1).

Well, that about wraps up the introduction. But where do we start with this journey?
The quantum phase (the smaller numbers) and the meta-phase (the bigger numbers) are explained separately, to prevent conflict when talking about each one. Without further ado, let's a-go!

Quantum Phase

Level 1 - The first quantum-entries

I already mentioned the y and x planes in the introduction. Its pretty simple when you get the pattern down. Let's go over y's prefixes and how they combine with x's numbers.

y prefix	Value	y prefix	Value
Mono-	1	Do-	2
Tro-	3	Tetro-	4
Pento-	5	Hexo-	6
Hepto-	7	Octo-	8
Enno-	9	Deco-	10
Hendeco-	11	Dodeco-	12
Triadeco-	13	Tetradeco-	14
Pentadeco-	15	Hexadeco-	16
Heptadeco-	17	Octadeco-	18
Ennadeco-	19	Icoso-	20
Icosiheno-	21	Icosido-	22
Icositro-	23	Icositetro-	24
Icosipento-	25	Icosihexo-	26
Icosihepto-	27	Icosiocto-	28
Icosienno-	29	Triaconto-	30
Tetraconto-	40	Pentaconto-	50
Hexaconto-	60	Heptaconto-	70
Octaconto-	80	Ennaconto-	90
Hecto-	100	...	...

Prefixes are used from 1-100. There are extra prefixes after this, but they are only optional since you can also do quantumnumbers like “Monero to Zero” which mean the same thing. The “o-” part is removed when combined with any normal number from 0-99 (unless otherwise specified).
At 100+, the whole thing is added on top of the number, including the hyphen.

For Zero (0):
Replace the “z” with the numbered prefix. They should appear like these:
Trero | q(3;0)
Tetrero | q(4;0)
Pentero | q(5;0)

For One (1) and numbers starting with “O” or “E”:
Just place the numbered prefix on top of the normal number. For the Mono- prefix, only add the “M” from the prefix. They should appear like these:
Trone | q(3;1)
Tetrone | q(4;1)
Pentone | q(5;1)
Treight | q(3;8)
Tetreight | q(4;8)
Penteight | q(5;8)

For numbers starting with “T”:
Mono- should have their “o” removed when combined. Do- has no necessary letters to remove. Tro- and Tetro- should have their “o” replaced with an “e”, for Tre- and Tetre-. Pento-, Hexo-, Hepto-, Octo-, Deco- through Ennadeco-, and Icoso- through Ennaconenno- (excluding ones that end with tro- and tetro-) should have their “o” removed when combined; also, if there are 2 “t”s in a row, remove one of the “t”s. Enno- should have the “no” part removed, for En-.
They should appear like these:
Tretwo | q(3;2)
Dectwo | q(10;2)
Trethree | q(3;3)
Decthree | q(10;3)

For numbers starting with “F”:
The rules for numbers starting with “T” apply here, with some modifications:
If a prefix after having their necessary ending letter removed doesn’t end with a “t”, “c”, or “s”, then the normal number’s beginning letter is “F”. If it does end with any of those letters, then the normal number’s beginning letter is with that letter.
They should appear like these:
Tetrefour | q(4;4)

Pentour | q(5;4)
Decour | q(10;4)
Tetrefive | q(4;5)
Pentive | q(5;5)
Decive | q(10;5)

For numbers starting with “S”:
Similar rules for numbers starting with “F”, with some modifications:
If a prefix after having their necessary ending letter removed doesn’t end with a “t”, “c”, or “x”, then the normal number’s beginning letter is “S”. If it does end with any of those letters, then the normal number’s beginning letter is with that letter.
For prefixes that end with “n”, if there are 2 ns in a row, there is no “s” for the normal number. If there is only 1 n, then there is a “s”.
They should appear like these:
Tetresix | q(4;6)
Pentix | q(5;6)
Hexix | q(6;6)
Tetreseven | q(4;7)
Penteven | q(5;7)
Hexeven | q(6;7)

For numbers starting with “N”:
Same scenario with zero (0), but instead you replace the “N” instead of “Z”.
They should appear like these:
Trine | q(3;9)
Tetrine | q(4;9)
Pentine | q(5;9)

Fortunately for you, if you do not want through the hassle of doing it yourself, I have an entire sheet that has these values done for you. Check it out here.
After Ninety-Nine, doing the same letter replacing thing won’t work here. To counteract this issue, I’ve created alternate names for numbers like One Hundred, Two Hundred, etc. You can find them in the same sheet I did all the values for you.

Most of these rules also apply to translations of these quantumnumbers, with extra rules for numbers starting with letters that are not found in the English translation.

For numbers starting with “U”:
Same as the other vowels (“O” and “E”).

Just place the numbered prefix on top of the normal number. For the Mono- prefix, only add the “M” from the prefix. The Spanish translations should appear like these:
Truno | q(3;1)
Tetruno | q(4;1)
Pentuno | q(5;1)

For numbers starting with “D”:
Same case for numbers starting with “T”, however for prefixes ending with an “s”, it is removed. For prefixes ending with a “c”, an extra “o” is added. The Spanish translations should appear like these:
Tredos | q(3;2)
Decodos | q(10;2)
Trediez | q(3;10)
Decodiez | q(10;10)

For numbers starting with “C”:
Same scenario with zero (0), but instead you replace the “C” instead “Z”. The Spanish translations should appear like these:
Truatro | q(3;4)
Tetruatro | q(4;4)
Pentuatro | q(5;4)

For numbers starting with “Q”:
Same scenario with zero (0), but instead you replace the “Q” instead “Z”, and for the Mono- prefix, only add the “M” from the prefix, and prefixes with “hen” at the end have the “en” part removed. The Spanish translations should appear like these:
Truince | q(3;15)
Tetruince | q(4;15)
Pentuince | q(5;15)

For numbers starting with “V”:
Same case for numbers starting with “T” but with some modifications.
Tro- and Tetro- keep their “o” instead of it being replaced with an “e”. Prefixes ending with these two also keep their “o”.
If a prefix after having their necessary ending letter removed doesn’t end with a “t”, “c” or “s”, then the normal number’s beginning letter is “V”. If it does end with a “t” or “c”, then the normal number’s beginning letter is that. If it ends with a “s”, then the normal number’s beginning letters are “sv”. If it ends with “nt”, then the “v” stays, even if it means replacing the prefix’s “t” but keeping the “n” intact. The Spanish translations should appear like these:

Tetroveinte | q(4;20)
Penteinte | q(5;20)
Deceinte | q(10;20)
Icosveinte | q(20;20)

One other thing to note, the y plane and the x plane have to appear in the notation. For instance, q(3) wouldn't be allowed since someone can just coin that for something else. Lower planes can be ignored, but these two are essential.

Level 1.5 - The z plane

Now let's look at a plane lower than the y plane: the z plane. The z plane also has it's own set of prefixes for 1-100. If the y plane is zero (0), then these prefixes have the same kind of rules as before. If the y plane is greater than or equal one (1), then the whole z prefix is attached to the y prefix.

z prefix	Value	z prefix	Value
Kua-	1	Shae-	2
Trau-	3	Fae-	4
Lyo-	5	Xae-	6
Thae-	7	Vae-	8
Noe-	9	Oru-	10
Kuoru-	11	Shaoru-	12
Traoru-	13	Faoru-	14
Lyoru-	15	Xaoru-	16
Thaoru-	17	Vaoru-	18
Novoru-	19	Surae-	20
Suraekua-	21	Suraeshae-	22
Auto-	30	Heto-	40
Delta-	50	Swepta-	60
Whorlo-	70	Arbito-	80

Specto-	90	Rusto-	100

There is an alternative for this. Add a matching y plane prefix and add it on top of "qua-". The three exceptions are: Mono- is Mon- (Monqua-), Tro- is Tri- (Triqua-), and all the rest after Tro- have the "o" replaced with a "a".
A value of q(2;3;0) means Shaetrero, q(15;11;3) means Lyoruhendecthree.
Both also mean Doquatrero and Pentadecaquahendecthree, respectively.

If the z plane is zero (0), then it is ignored in the notation. q(0;1;0) = q(1;0).

Level 2 - 4 and more planes?

Now, let's try and shake things up for the small numbers. What happens when you add a 4th plane into the equation? 5 planes? 6 planes?
There's no unique prefixes for the 4th plane and up, but the same y plane prefix matching is still possible here. Like how "qua-" is a stem prefix for the 3rd plane, we can add more to higher quantum planes. Here's how they would follow:

Stem Prefix (v1)	Plane	Stem Prefix (v2)	Plane (uses French)
qua-	3	zwe-	2
quo-	4	dre-	3
ka-	5	vie-	4
ko-	6	fun-	5
kra-	7	sec-	6
kro-	8	sie-	7
tra-	9	ach-	8
tro-	10	neu-	9
tha-	11	zeh-	10
tho-	12		
dra-	13		
dro-	14		
gla-	15		

glo-	16		
gra-	17		
gro-	18		
sha-	19		
sho-	20		
ska-	21		
sko-	22		
sca-	23		
sco-	24		
cla-	25		
clo-	26		
xuna-	27		
xuno-	28		
ruta-	29		
ruto-	30		

Some examples include:

1. q(2;0;1;1) = Doquomone
2. q(10;0;2;0;3;3) = deckodoquotrethree
3. q(32;12;0;0;0;3;0;3;10) = triacondotradodecakrotriquotreten

Also, same rule from the z plane: if there are multiple zeros in a row, where there are no planes that have a value greater than or equal one, then they are ignored. If all quantum planes are 0, then leave one left.

q(0;0;1;2) = q(1;2)
q(0;0;2;0;0;3) = q(2;0;0;3)
q(0;0;0;1) = q(0;1)
q(0;0;0;0) = q(0;0)

Level 3 - Nesting planes
Now, it is cool to have many different planes that are smaller than each other, but what if we wanted to take it to the extreme? Well, this is where we meet [n].
[n] counts how many planes that have the same value as the value that is next to this on the right. Examples:
q(1;0[2];1) = q(1;0;0;1)
q(1;0[3];1) = q(1;0;0;0;1)

[n] can be applied to any of the planes, but will not work with the rightmost entry, which is the x plane. We have something different for that plane.
You can even chain multiple of these in a row, like:
q(1[3];3[2];2[5];10) = q(1;1;1;3;3;2;2;2;2;2;10)

These nests can also get multiple entries through the same method you increment a plane by one from an earlier plane. So q(1;0[□];1) turns into q(1;0[1;0];1).

There is a semi-colon after each nest, as something like q(2;3[2]4) may be confusing on whether we're still in the quantum phase or not.

Level 4 - Planes inside nests
Let's talk more about multiple entries in nests from the previous level, shall we? So after getting q(1;0[1;0];0) from q(1;0[□];0) (x isn't affected), where do we go from here?

Meta-Phase
Level 1 - The first meta-entries
We've already talked about how any of the planes that reach eternity (□) reset to 0 and increment the next entry in the right by one. But, what happens when the x plane reaches eternity? Well, we start to see the introduction of meta-entries. (not to be confused with any numbers related to meta-numbers, meta-entries, etc.)

To distinguish between planes and meta-entries, we use commas after x. Only the x plane can utilize commas in this manner.
q(0;□) = q(0;0,1)

As you can see, we can effectively generate much higher level planes beyond our normal number plane. In this case we've made x_1 after having the x plane (or x_0 plane) reach eternity, and all of these new planes can also be made in the same way.
q(0;0,□) = q(0;0,0,1)

What kind of name do we get at eternity? Dumune. This is a new level of “one” beyond all numbers in standard fictional googology, and every new plane surpasses a new “FxG” scale (if you know you know).

The “Du-” prefix can be incremented with either each 1,000th illion prefix, like:
Du-, Tru-, Quadru-, Quintu-, Sextu-, Septu-, Octu-, Nonu-, Decu-, etc.
Or the prefixes we’ve been using for the y plane (with the exception being the “o” being replaced with a “u”), like:
Du-, Tru-, Tetru-, Pentu-, Hexu-, Heptu-, Octu-, Ennu-, Decu-, etc.
The “-mune” suffix can be individually incremented with each number in the prefix’s respective plane, using the illion prefixes.
Examples:
Dumune = q(0;0,1)
Dubune = q(0;0,2)
Dutrune = q(0;0,3)
Duquadrune = q(0;0,4)
Trumune = q(0;0,0,1)
Trubune = q(0;0,0,2)

For equations like q(0;1,1,1), you essentially chain them with hyphens like so:
Trumune-One = q(0;1,0,1)
Trumune-Two = q(0;2,0,1)
Trumune-Dumune = q(0;0,1,1)
Trumune-Dumune-One = q(0;1,1,1)
The highest numbered word goes first, then goes down in planes, ignoring any zeros (if any are in the array).

You can even combine this with the quantum planes established in the quantum phase of this notation.
Mono-Dumune = q(1;0,1)
Do-Dumune = q(2;0,1)
Kua-Dumune = q(1;0;0,1)
Kuamono-Dumune = q(1;1;0,1)
Mono-Trumune = q(1;0,0,1)
Mono-Trumune-Dumune = q(1;0,1,1)

If there are any leading zeros, ignore entries that don’t have a value higher than or equal one up to x_0.
q(0;0,1,0,0) = q(0;0,1)
q(0;1,3,0,0,3,0) = q(0;1,3,0,0,3)
q(0;0,0,0) = q(0;0)

Level 2 - Nesting meta-planes

Now, you may be wondering how we can represent numbers like Centumune without typing a whole bunch of zeros. That's when we introduce {n}.
{n} counts how many planes that have that same number as the one on the left of it. For example, q(0;2{10}) means there's ten copies of "2" in the array, which is q(0;2,2,2,2,2,2,2,2,2,2).

If the number on the left of {n} is zero, the leading zeros apply and the nests cancel out.
q(0;0{100}) = q(0;0)
To prevent this, numbers on the right of {n} are ignored. This means you can have values like q(0;0{100}1) and it will not turn into q(0;0,1).

Nests that look identical can be grouped together. For instance, q(0;0{10}0{25}1) can be grouped into q(0;0{35}1). You can just add the two numbers in {n} together. If there's even a small difference like q(0;0{10}1{25}2) is present, they can't be added together. They have to be the exact value for the nests to be combined.

The {n} type nest can only be applied to the x plane. Arrays like q(2{3}4{210}4;3{200}) are considered invalid. There is also no comma after each nest, since the y plane is always visible, so there's no need for an indication that we're in the meta-phase of the notation.

Level 3 - Meta-planes in nests

The next natural step is adding multiple entries in these nests. When {n} reaches □, like q(0;0{□}1), we upgrade to q(0;0{0,1}1).
These planes can then increment just like the ones outside of nests. To upgrade to q(0;0{1,1}1), we first need to reach q(0;0{0,1}2), which we can do by doing a nest next to the recursion one. So, q(0;0{3}{0,1}1) = q(0;0,0,0{0,1}1). Once we reach q(0;0{□}{0,1}1), we can then enter q(0;0{0,1}2), to which we can finally get q(0;0{1,1}1) from q(0;0{0,1}□).

We can also stack nests inside of nests, to which we can get to q(0;0{0{0,1}1}1) from q(0;0{0{□}1}1). We can keep stacking the nests until we get to the first operator of this phase!

Level 4 - The first plane operator

Now, we technically can keep stacking it forever, but that isn't nearly as strong as condensing it into an operator. That's when we introduce : .
Let's say we want to condense q(0;2{2{2{2{2,3}3}3}3}3). : will tell you how many of x_0 and (x_0+1) in the curly brackets there are, almost like how operator notation works.

This is shown as q(y;x_0{$x_{0,1}$:$x_{0,2}$}). In this state, $x_{0,1}$ must always be greater than or equal to one (1), otherwise it'll become q(0;0).

For example, q(0;2{5:1}1) is a condensed version of q(0;2{2{2{2{2{2,3}3}3}3}3}3). As you can see, the right inner number and the numbers at the right are +1'd instead of all the planes being the same. This is because if all of them were the same number, they'd all be condensed instead of being separate, but also if there is a zero, then it would just become q(0;0) again, and we don't want that!

Entries on the right of this will be added at the end of the array instead of being added to the nests, so q(0;2{5:1}2) becomes q(0;2{2{2{2{2{2,3}3}3}3}3}3,2).

If $x_{0,1}$ reaches eternity (□), it nests like normal. q(0;0{□:1}1) becomes q(0;0{0,1:1}1). The : operator can be nest-stacked like q(0;0{0{0{0{0{0,1:1}1:1}1:1}1:1}1:1}1). Except for x_1 in the far right, all of the nests have ":$x_{0,2}$" in the right of each bracket. $x_{0,2}$ will increment when we do this;
q(0;0{0{0{0{0{0,1:1}1:1}1:1}1:1}1:1}1) = q(0;0{5:2}1). Like $x_{0,1}$, if $x_{0,2}$ reaches eternity (□), it also nests like normal, so q(0;0{1:□}1) becomes q(0;0{1:0,1}1).

We can get multiple : operators in play by nest-stacking like this:
q(0;0{1:0{1:0{1:0{1:0{1:0,1}1}1}1}1}1) = q(0;0{5:0:1}1)
Except for x_0, all the nests have "1:" at the left of each nest, since these are $x_{0,1}$'s.
Now we have $x_{0,1}$, $x_{0,2}$, and $x_{0,3}$ in play. All of these new planes have the same attributes as before.
We can keep going to get $x_{0,4}$, $x_{0,5}$, etc.

There can also be nests in the operators themselves. There's two ways to display this:
First method: use subscripts. q(0;0{0$:_{\{9\}}$1}1) = q(0;0{0:0:0:0:0:0:0:0:0:1}1)
Second method: add a forward slash between {x} and : .
q(0;0{0:/{9}1}1) = q(0;0{0:0:0:0:0:0:0:0:0:1}1)

Now, can you see where we're going with this? We can stack these operator-nests like we did in prior sections. Stacking these gives us :: , which means these operators can also be layers by stacking them like ::::: .
First, how is :: reached? Pretty simple, q(0;0{6::1}1) = q(0;0{0:/{0:/{0:/{0:/{0,1}1}1}1}1}1).
$x_{0,2}$ at the right of :: increments as q(0;0{6::2}1) =
q(0;0{0:/{0:/{0:/{0:/{0,1::1}1::1}1::1}1::1}1::1}1).
This can keep going to ::: , :::: , ::::: , etc. until we hit *another* nest that counts colons (:).

There's again two ways to display this:

First method: use superscripts. q(0;0{0:^{10}1}1) = q(0;0{0::::::::::1}1)
Second method: add a backward slash between {x} and : . The curly brackets for this is changed out for square brackets.
q(0;0{0:\[10]1}1) = q(0;0{0::::::::::1}1)

Now, this is when we finally get to the next operator after stacking these nests.

Level 5 - The second plane operator

For our next operator, we'll be using $. Unlike before, $ doesn't count the ones outside the curly brackets, so x_1 at the right can be ignored now. (except for getting colons (:) in here)
q(0;0{5$1}1) = q(0;0{0:\[0:\[0:\[0:\[0,1]1]1]1]1}1)

▯ / ☍ [planned for later levels]

Post Meta-Phase

There's some crazy things that happen after going very far into the notation.
Let's take a look at them!

Level 1 - Iteration building

This is the way we're going to build beyond everything we've built
What we've been building is only the first iteration of the notation, denoted as q(...;y;x,...|m^1).
m^1 in this case is denoted as 1. q(4;3) is the same as q(4;3|1). The "|1" is not that important for the meta-phase, but once we get to iteration building it's important to apply the "|1" for embeds inside the outer structures.

After ????? (the exact point will replace this placeholder as the Meta-Phase is finished), you will reach q(0;0|2). Immediately things are different even if they're under the hood. That is, until you get to Eternity in one of the planes.

You will notice that q(▯;0|2) doesn't actually reset into q(0;1|2). That is because the value for when the reset happens is changed to a much higher one. While ▯ itself might not necessarily reset and increment the next plane, you do still want to change that into q(0;0,1|1) in order to keep incrementing the plane *until* the plane does reset. So q(▯;0|2) = q(q(0;0,1|1);0|2).

Also, I haven't explained what happens when there is an embedded structure inside a structure, so let me explain this:
As long as the iteration number in the embedded structure is less than or equal to the outer structure, like q(0;q(0;0,1|1),1|2) and q(0;q(0;0,1|2),1|2), then you are

allowed to do so. q(0;q(0;0,1|3),1|2) is not allowed since the embedded structure's iteration is bigger than the outer one like q(0;q(0;0,1|m^1+n),1|m^1).

Here's where the iterations reset at:
Iteration one planes reset at Eternity (□) or q(0;0,1|1). q(□;0|1) = q(q(0;0,1|1);0|1) = q(0;1|1)
Iteration two planes reset at q(0;0,1|2). q(q(0;0,1|2);0|2) = q(0;1|2)
Iteration three planes reset at q(0;0,1|3). q(q(0;0,1|3);0|3) = q(0;1|3)
etc.

Post Quantum Notation

After taking the notation to it's absolute limit, you will notice some similar cardinals start appearing beyond this notation. Let's look at a couple of them! [HUGELY WIP]

Level 1 - ?????

?????

Cosmetic Properties

Now that we have gone through both phases of this notation, let's look at other properties this notation offers that may be in interest to you.

Property 1 - Base-100 operations

So, we know for a fact that any number between zero and eternity can be used here. Let's look at some of the usable values here.

As you know, in my Meta-Quantum Googology Adventure series, my number system is base-100. But this notation uses base-10, so how can we display those base-100 operations in here?

There's no need for additional brackets and such for numbers like 10^^10, 10{10}100, etc.
For base-100, surround the number with <>s. This should allow you to use all the values from this doc in this notation, as long as it is inside these brackets.

These base-100 numbers can be used in any plane, even the meta-entries from the meta-phase of this notation.

Examples:
q(<10::10>;3)
q(3;2,<10{:}25>,100)
q(100;0{<24::10>}3)

Property 2 - Plane-Level Addition

Someone may ask if you could say add q(1;0;3,1) with q(2;3,3,4), or multiply q(3;2) by q(2;3;2,4,5), and I say YES!

Let's look at each of these operations in greater detail...
Plane-Level Addition:
We'll use a more simpler example for this.

Say we want to add q(0;2) with q(1;0,1) together?
What we need to do is look at the planes for both numbers, and match them up.

```
  q(0;2)
+q(1;0,1)
========
```

You can add extra zeros to the left or right if necessary, like so:

```
  q(0;2,0)
+q(1;0,1)
========
```

From there, just add each of the respective planes individually.
y plane: 0 + 1 = 1
x_0 plane: 2 + 0 = 2
x_1 plane: 0 + 1 = 1
So, the entire expression becomes q(1;2,1).

Let's do a more extreme example.
Say we want to add q(2;0;3;4,3{3}2) with q(1;3;2;3;2,4{4}2) together?
Now we're dealing with nesting planes here. Since the numbers in these nests are small, we can technically expand them, but I'll show a more extreme example with that.
With additional zeros and the nests expanded, we have this expression:

```
  q(0;2;0;3;4,3,3,3,2,0)
+q(1;3;2;3;2,4,4,4,4,2)
==================
```

From here, just add up all the planes individually like before. You should get an answer of q(1;5;2;6;6,7,7,7,6,2). We can then condense this into q(1;5;2;6;6,7{3}6,2).

Now let's try to add q(0;3{100}1) with q(0;4,0{80}1). Now, if you want to be crazy, you could expand these and have to deal with hundreds of digits, but we're not going to do that here.

If you can figure it out, you'll find out that the first expression has 101 meta-entries, and the second has 82 meta-entries.

```
  q(0;3,3{99}1)
+q(0;4,0{80}1)
============
```

We know the number after the nest is one for both, which we need to keep account of while adding these up. First, add up the non-nested entries on the left. Only add the planes that are not next to the nest and are not after the first nest.
y plane: 0 + 0 = 0
x_0 plane: 3 + 4 = 7
STOP HERE!

Our next step is figuring out the rest of the operation. Since 3 + 0 is still 3, we can safely add the 80 0s with the 80 3s. This becomes 3{80}. Current expression: q(0;7,3{80}...
The nest entry to add is 3 + 1, which stopped us from just adding the whole thing. The result is 4. Current expression: q(0;7,3{80}4...

Since this is where the 2nd expression ends, we can add in the rest of the first expression on top. If you think about it, we subtracted 80 from 99, which became 19. We then added one entry with another, so it became 18. We can then add 3{18}1 to the rest, so the result becomes q(0;7,3{80}4,3{18}1). The first expression has 101 entries, and this result is 101 entries, so it works!

This is as far as it goes, though. But adding smaller nested arrays surely has to be an accomplishment, right?

An honerable mention: If you're adding a normal number, like 13, with an array, the normal number is treated as x_0, so q(2;4,2)+10 = q(2;14,2)

Property 3 - Structural Addition

We just did plane-level addition in our last property, but what about a different way to add arrays? Well, this uses ⊕ as the operation's symbol.

Structural addition is different from plane-level addition. Instead of adding entries at each plane level like plane-level addition, structural addition merges the inner architectures of two q(...z;y;x_0,x_1,x_2...) structures. What this does is:

- Combines layers, brackets, recursions, and compressions
- Can preserve plane length differences (unequal planes)

- Supports meta-structural behaviors, like pattern expansions or deep nesting fusion

Structual Addition appends all planes from the second structure after the first structure’s planes. For example:
q(3;2;5) ⊕ q(11;3;9) = q(3;2;5;11;3;9)
Think of this as:
$q(z^1;y^1;x^1) \oplus q(z^2;y^2;x^2) = q(z^1;y^1;x^1;z^2;y^2;x^2)$ [which becomes $q(x_{-5};x_{-4};x_{-3};x_{-2};x_{-1};x_0)$]
If the length of both structures mismatch, then add missing plane entries with zeros to align them before appending.
If there are meta-planes as well, then they are appended to this:
$q(z^1;y^1;x^1{}_0,x^1{}_1,x^1{}_2) \oplus q(z^2;y^2;x^2{}_0,x^2{}_1,x^2{}_2) = q(z^1;y^1;x^1{}_0;z^2;y^2;x^2{}_0,x^1{}_1,x^1{}_2,x^2{}_1,x^2{}_2)$

If one side of any or both structures have quantum-nests / [n] or meta-nests / {n}, expand them temporarily, append all planes, then condense them later.

Meta-Nest Example:
q(1;0{3}1) ⊕ q(0;0,0,2)
Expand the first structure so it becomes:
q(1;0,0,0,1) ⊕ q(0;0,0,2)
Now you can append them to get q(1;0;0,0,0,1,0,0,2), then condense to get q(1;0;0{3}1,0{2}2).
Quantum Phase and Meta-Phase are appended in their own sides, not together. This is why it is q(1;0;0,0,0,1,0,0,2) and not something like q(1;0,0,0,1;0;0,0,2).

Quantum-Nest Example:
q(1;0[3];1) ⊕ q(0[3];2)
Expand both structures so it becomes:
q(1;0;0;0;1) ⊕ q(0;0;0;2)
Now you can append them to get q(1;0;0;0;1;0;0;0;2), then condense to get q(1;0[3];1;0[3];2).

This method only works for smaller values of {n} and [n] respectively. But, it’s only to make it easier on yourself. You can append them as is without expanding them.
q(1;0[3];1) ⊕ q(0[3];2) would result in q(1;0[3];1;0[3];2) just the same way!
For the meta-nest example:
q(1;0{3}1) ⊕ q(0;0,0,2) = q(1;0;0{3}1,0,0,2) = q(1;0;0{3}1,0{2}2)

Recursions work the same way in this regard.
q(0;2{1,1}1) ⊕ q(0;3{32,21}1) = q(0;2{1,1}1,3{32,21}1)

If the 2nd value is a normal number, like 20. It is interpreted as a structure and appended at the end of the last active plane. For instance:
q(3;2;5,4{2}4) $\oplus$ 30 = q(3;2;5,4{2}4,30)
q(2;2{3}) $\oplus$ 3 = q(2;2{3}3)

Property 4 - Plane-Level Multiplication

If plane-level multiplication is adding each matching entry together, then plane-level multiplication would naturally be multiplying matching entries on each plane.
Let's multiply q(2;3) by q(3;5;2), shall we?
The first equation is shorter than the second. In this case, we treat missing entries as 1 (the identity for multiplication).
q(1;2;3) x q(3;5;2) becomes q(3;10;6). Let's break this down:
z plane: 1 x 3 = 3
y plane: 2 x 5 = 10
x_0 plane: 3 x 2 = 6

Now let's do q(2[3];4,3) x q(2;5,3{4}2).
I'll also show you how these can be multiplied without expanding, but for now let's expand these for this example.

q(2[3];4,3) = q(2;2;2;4,3)
q(2;5,3{4}2) = q(2;5,3,3,3,3,2)
Again, for this operation we'll treat missing entries as 1.

```
  q(2;2;2;4,3,1,1,1,1)
x q(1;1;2;5,3,3,3,3,2)
=================
```

Now just multiply them with normal multiplication rules. You should get a result of q(2;2;4;20,6,3,3,3,2). We can condense this into q(2[2];4;20,6,3{3}2).

Now for the extreme example again. Let's multiply q(3;4{100}2) by q(2;3{85}4,7). The first expression has 101 meta-entries, and the second has 87 meta-entries.

The first thing to do is check for alignment. Since there's 87 meta-entries in the second expression, the missing 14 at the end will be interpreted as ones.

Next, multiply the non-nested planes together. In this example, only the y plane can be multiplied right away, so 3 x 2 = 6.

Next, notice the nests in both expressions. We can multiply the first 85 4s from the first array with the 85 3s in the second. 3 x 4 = 12, so all of these become 12. Current expression: q(6;12{85}...

Now, the next two 4s from the first array can be multiplied with the remaining two meta-entries in the second array. 4 x 4 = 16, 4 x 7 = 28. Now add this to the result. Current expression: q(6;12{85}16,28...

The rest of the first array can be multiplied by ones. In other words, adding them on top of the result. The final result is q(6;12{85}16,28,4{13}2). It has 101 meta-entries like before. Nice!

Same thing for plane-level addition, if the notation is being multiplied by a normal number like 20, it will treat the normal number as x_0. So q(0;2,3) x 3 = q(0;6,3). This will also apply to future plane-level operations, which we'll talk about down below.

Property 5-7 - Other Plane-Level Operations

Higher level operations like exponentiation, tetration, etc. along with factorials, basically apply to all planes like addition and multiplication do. Here are brief explanations for these:

Property 5: Plane-Level Exponentiation
All the planes are individually rised to the power of matching planes.
Missing planes are treated as ones, like plane-level multiplication.
q(4;3,7,2)↑q(2;3;2,4,2,8) =
q(1;4;3,7,2,1)↑q(2;3;2,4,2,8) = q(1;64;9,2401,4,1)

Example with nests:
q(1;3{20}4{13}2)↑q(0;4{30}3)
First, rise the y plane numbers, in this case:
$1^0 = 1$
Then, rise the 20 3s from the first array by the first 20 4s from the second array like this:
$3^4 = 81$
Then, rise the first 10 4s from the first array with the last 10 4s from the second array like this:
$4^4 = 256$
Now, rise the next 4 from the first array with the 3 from the second array:
$4^3 = 64$
Finally, rise the remaining entries from the first array with ones.
The final result should be q(1;81{20}256{10}64,4{2}2).

Property 6: Plane-Level Factorials
The factorial applies to each plane entry. This is one of the exceptions that will apply to all planes, regardless of whether its normal factorial, subfactorial, etc.
q(10;8,13)! = q(10!;8!;13!) = q(3628000;40320,6227030800)
This grows really fast but pales in comparison to hyperoperations like tetration.
This doesn’t apply to nests. For example, q(0;2{3}3)! ≠ q(0!;2!{3!}3!).

Property 7: Plane-Level Hyperoperations
Same rules apply, match planes by position, and treat missing entries as ones.
q(2;3,4)↑↑q(2;2,3) = q(4;27,4^256)
q(2;2;3,3)↑↑↑q(2;3,3,4) = q(2;2;3,3,1)↑↑↑q(1;2;3,3,4) = q(2;4;$3\uparrow\uparrow3^{27}$,$3\uparrow\uparrow3^{27}$,1)
These still scale like they do with normal numbers, so careful use is a necessity!

Property 8 - Plane-Level Subtraction

Now, if we have plane-level addition, plane-level multiplication, etc., it would make sense to have plane-level subtraction, right?

It may sound simple enough, I mean, you just subtract the higher values in each plane by the lower values in each plane, and it works, right? Hehe, WRONG!
There are many complications that make this more difficult to gauge than before. We need to define not just nests and large number stuff like googology does. Now this also needs scenarios like if a plane hits negative.

Now, we could use the borrowing trick like what we do for specific numbers, like 300-123. However, if you remember correctly, recursions happen after a plane hits eternity, and we can’t just subtract eternity by a normal number. So instead, we’ll *have* to allow negative values in planes to make this work. This opens up more possible properties to this notation. More on that later.

Since we’re not going to do complex borrowing to prevent negatives, **we’ll have to allow negatives to exist to make this operation work.**
For instance, q(3;2,43)-q(2;3,50)
Since we’re allowing negatives in this array, we can just subtract each plane separately freely, even if it results in negatives.
So q(3;2,43)-q(2;3,50) = q(1;-1,-7).

In instances like q(10;0,3)-q(3;4;0,4), the missing planes are interpreted as zero.
So q(10;0,3)-q(3;4;0,4) = q(0;10;0,3)-q(3;4;0,4) = q(-3;6;0,1)

For a more extreme example, say we want to subtract q(20;0,4{30}3{20}1) by q(11;3;5,2{45}5).

The first array has 1 quantum plane and 52 meta-entries, and the second has 2 quantum planes and 47 meta-entries.

First thing's first, align both arrays with missing zeros.
  q(0;20;0,4{30}3{20}1)
- q(11;3;5,2{45}5,0{5})

Let's first subtract the non-nested planes.
z plane: 0 - 11 = -11
y plane: 20 - 3 = 17
x_0 plane: 0 - 5 = -5
STOP HERE!

Next, subtract the 30 4s by the first 30 2s. 4-2 = 2. Now subtract the first 15 3s with the remaining 15 2s. 3-2 = 1. Now subtract the next 3 by 5, which is -2. Finally, subtract the rest of the first array with zeros, which stays the same.
So the final result is q(-11;17;-5,2{30}1{15}-2,3{4}1).

Property 9 - Plane-Level Division + n-Roots

The next challenge is plane-level division. Well, not really. Rationals are also allowed in any plane (not for nestings), so dividing is possible with no stress.
Like plane-level multiplication, plane-level division treats missing planes as ones.
q(3;2,5) / q(2;4;3,2) = q(1;3;2,5) / q(2;4;2,2) = q(0.5;0.75;1,2.5)

You can do this with {n} nests and [n] too! Just retrace the steps we've taken for the other plane-level operators, and you're golden!

Now, I know what you are going to ask: what happens if at least one plane in the divisor array is zero?
Pretty simple: in the plane where the [CANNOT DIVIDE MY ZERO] error occurs, the resulting plane shows the ⌽ symbol, and the entire array becomes invalid.
For example:
q(1;3;2) / q(1;0;2) = q(1;⌽;1) = INVALID
Two or more planes experience the error:
q(1;0;3,4,1) / q(3;1;0,3,0,1) = q(0.333...;0;⌽,1.333...,⌽,1) = INVALID

This is unfortunately as far as plane-level operations can go. n-Roots would apply to all planes at once, even if the multiplier/divider is a normal number.

Property 10 - Structural Multiplication

If structural addition is appending two structures together, then what does structural multiplication do? [The symbol for this is ⊗]
Well, trying to use two structures will not work here, but we can use a structure and a normal number to make this work.
Structural Multiplication implies repetition of the same structure. It does not multiply the numbers inside, but rather repeated the structural layout of planes as a larger composite structure. For example:
q(0;3,4) ⊗ 2 means copying the “3,4” structure 2 times, so it becomes q(0;3,4,3,4).
This applied to both phases. An operation of q(0;3,5{20}4) ⊗ 3 copies the “3,5{20}4” structure 3 times, so it becomes q(0;3,5{20}4,3,5{20}4,3,5{20}4).

Property 11-12 - Structural Subtraction + Division

Structural Subtraction removes planes that have exact matching values and matching plane value. This uses the ⊖ symbol.
Some other rules include:

- If one plane’s number doesn’t match the other, you skip it, remove the plane from the second plane, and you go to the next plane.
- If all the planes don’t have matching numbers, then nothing happens.
- If y has matching numbers and is the only quantum plane, then y returns zero (0).
- If removing leaves an empty structure, it returns as zero (0) with no structure.

Let’s look at q(3;4,5) ⊖ q(3;5,5). Since the threes turn into zero, that’s already our plane done. The x_0 plane has no matching numbers, so that is skipped. The x_1 plane does have matching numbers, so this plane gets removed. This leaves us with q(0;4).

For operations like q(5;3,2,1) ⊖ q(4;5,3), since none of the planes match up, the second plane disappears, which leaves us with q(5;3,2,1).

Structural division is the inverse of structural multiplication, using the ⊘ symbol. This breaks down structures into smaller units. Some rules are:

- If dividing by a scaler (like 20 outside a structure), then break down the structure’s copies of those planes by that. If the planes are not even, then it becomes invalid.
- If dividing by a structure, try to break down the left structure into groups that match the right structure’s size, but if the right structure isn’t a valid divider, then it is invalid.

Let’s look at q(6;6;6;6,6,6,4,4,4) ⊘ 3. Rule 1 applies here, so we break this down to q(6;6,4). If there’s even one plane not equal, like q(6;6;6;6,7,6,4,4,4) ⊘ 3 or q(6;6;6;6,6,4,4,4) ⊘ 3, then this becomes invalid.

For q(6;6;4,4,2,2) ⊘ q(6;4,2), rule 2 applies, so look at the planes and scale down the structure to how many copies there are. There are 2 copies of 6s, 4s, and 2s, so the answer becomes 2. If there’s a mismatch like q(6;6;4,4,2,2) ⊘ q(6;5,2) or q(6;6;4,2,2) ⊘ q(6;4,2), then this becomes invalid.

Property 13-16 - Structure-Scaler Operations

So, when we looked at plane-level multiplication and plane-level division, you know that if an array is being multiplied/divided by a normal number (like 30), it only affects the x_0 plane. But, what if we had a variant of those operations where a normal number affects all active planes at once? This is called structure-scaler operations.

Let’s look at some operations that support this variant. Addition and Subtraction don’t support this version.

Property 13: Structure-Scaler Multiplication
For this variant, we need to use ⋆ for our multiplication symbol.
Let’s look at q(3;2;4,4) ⋆ 2. Since the 2 affects all planes here, let’s multiply each plane one-by-one.
3 x 2 = 6
2 x 2 = 4
4 x 2 = 8 [two planes]
So the result becomes q(6;4;8,8). Nice!

Let’s look at a more extreme example: q(3[10];4[20];5{25}4) ⋆ 10. Since the second value doesn’t affect nests, all we need to do is multiply each of the planes by 10, including nests.
3 x 10 = 30 [nested 10 times]
4 x 10 = 40 [nested 20 times]
5 x 10 = 50 [nested 25 times]
4 x 10 = 40
So the final result is q(30[10];40[20];50{25}40).

Also, because this only affects the values outside of nests, we can even use this on extreme nested arrays like q(0;30{2,5,3}5{3,2,1}4). This array structure-scaler multiplied by 25 results in q(0;750{2,5,3}125{3,2,1}100). Amazing!

One more thing, since the second number affects *active* planes, no additional planes are added to the result.

Property 14: Structure-Scaler Division
This variant of division uses the ⋇ symbol.
Let’s look at q(20;25,18) ⋇ 2. Same thing for the multiplication variant, the 2 affects all active planes, so this results in q(10;12.5,9).

So, what we discussed about structure-scaler multiplication also affects structure-scaler division. Only affects values outside of nests, so small nests and large nests aren’t affected.

Property 15: Structure-Scaler Exponentiation + Hyperoperations
These variants uses the ⇑ symbol. Hyperoperations use the same amount of ⇑ as they do for normal arrows.
q(2;3,5) ⇑ 2 becomes q(4;9,25).
q(2;3,4) ⇑⇑ 2 becomes q(4;3^27,4^(1.34x10^154))

Property 16: Structure-Scaler n-Roots
The root function, by default, affects all planes, so plane-level n-roots are non-existent.

Property 17 - Negative Numbers
Negative numbers can exist in each plane. The negativity is associated with the plane they are in, not the whole number. This means that naming must reflect this.

With positives, q(3;2) is Tretwo. If y is negative, in this case q(-3;2), we need a little prefix to adapt this system. We can use the Neg- prefix for the negative planes. This becomes Negtritwo. If the x_0 plane is negative, then the initial rule for numbers greater than or equal one hundred applies, but instead the reason is for the negatives. So q(3;-2) is Tro-Negtwo.

Just like how whenever a plane hits eternity (□), that plane resets to zero and increments the next plane to the right by one, when a plane hits *negative eternity* (-□), that plane resets to zero and *decrements* the nest plane to the right by one. So when q(-□;0) happens, it will turn into q(0;-1).

A negative sign can still be put on the outside of the notation, making all planes negative.

Property 18 - Complex Numbers

Complex numbers like 2+2i are treated as raw values in the plane they are in, just like rational and negative numbers. These now make the notation feel more like a multi-dimensions lattice of values across real and imaginary axes. Now it’s no longer a notation that just shows larger and smaller versions of normal numbers inside each other, there’s a lot of complex dimensions within dimensions!

This was originally something I wanted to do for 2025’s April Fools joke, but now with a notation I can visualize this with, it’s literally reality!
Now, how does this affect the operations for this notation?
Plane-Level Addition: Add corresponding complex values per plane.
Structural Addition: You still append structures as usual, whether there’s complex numbers in one structure’s planes or both.
Plane-Level Multiplication: Multiply each matching plane using complex algebra rules.
Plane-Level Exponentiation: Matching planes are rised with standard complex exponent rules. Results will be complex, and sometimes have multi-values in the planes.
Plane-Level Factorials: Standard factorials don’t work with complex numbers, but using the Gamma function will allow non-integers and imaginary values to work.
Plane-Level Hyperoperations: Matching planes using hyperoperations work, though results will get very complex.
Plane-Level Subtraction: Subtract corresponding values, respecting complex algebra. Negatives are allowed, and so is it allowed as a valid complex number. No borrowing needed.
Plane-Level Division: Divide corresponding values like before. Division by zero is still the same; any plane with a denominator of 0 or 0+0i becomes ⌽ and invalidates the entire notation.
Structural Multiplication: Copying structures works like normal, whether there’s complex numbers in any of the planes or not.
Structural Subtraction:
Structural Division:
Structure-Scaler Multiplication: Multiply each plane by scaler normally.
q(2;3+2i,3) ⋆ 2 results in q(4;6+4i,6).
Structure-Scaler Division: Divide each plane by scaler normally.
q(20;32+42i,19) ⋇ 2 results in q(10;16+21i,9.5).
Structure-Scaler Exponentiation + Hyperoperations: Works like normal, affects all values, even complex ones.
Structure-Scaler n-Roots: Works normally for real numbers, but complex numbers give out multiple solutions. This happens because complex numbers have an angle (argument) that gets split into multiple parts. So what we need to do is:

First, convert to polar form. Any complex number z = a+bi can be written in polar form as:
$z = re^{i\theta}$
Now apply the n-th root. The formula for this is:
$\sqrt[n]{z} = \sqrt[n]{r} \text{ x } e^{i(\theta+2\pi k)/n}$
for k = 0,1,...,n-1
Depending on n, and how many planes have complex number, you can get multiple possible structures.

When an imaginary number reaches eternity (□), the next plane increments an imaginary number by one instead of a real number, so q(0;2+□i,2) = q(0;2,2+i) or q(0;2,2+1i).

Property 19 - Imaginary Structures?

Just like how the planes in real structures can have imaginary/complex numbers in them, there can be imaginary structures as well! They have the imaginary i at the outside of a structure, making it imaginary. They behave just as you expect. Adding and subtracting imaginary structures work like real structures do.
q(3;4,2)i + q(2;3,5)i = q(5;7,7)i
q(10;4,11)i - q(3;7,8)i = q(7;-3,3)i
Since i times i, or i squared, results in -1, multiplying two imaginary structures gives us a real structure, so q(3;4,2)i x q(2;1,3)i = q(-6;-4,-6).
Multiplying a read structure by an imaginary structure keeps it imaginary, so q(4;2,5) x q(3;7,4)i = q(12;14,20)i.

Imaginary structures behave similarly to real structures. Most operations, including structural operations, work as they should. If special rules are necessary, then use them while solving them. Now let's look at complex structures.

Complex structures follows the same logic as normal complex numbers, excepts now each component is a structures values instead of a simple number. Adding and subtracting has the real and imaginary parts are done separately, like normal complex numbers.
[q(4;2,5)+q(5;3,11)i] + [q(3;5,4)+q(2;6,21)i] = [q(7;7,9)+q(7;9,32)i]
[q(50;40,81)+q(40;31,24)i] + [q(23;34,21)+q(71;32,99)i] = [q(27;6,60)+q(-31;-1,-75)i]
Multiplication follows the FOIL rule like how (a+bi)(c+di) works for normal complex numbers.
WIP

Now, structural addition is where we need distinction between real and complex structures. Now something like q(3;2,5) ⊕ q(4;3,2) would become q(3;2;4;3,5,2), but

for something like q(3;2,5) ⊕ q(4;3,2)i, they can’t append. For cases like this, the ⊕ becomes a normal plus sign, so it becomes q(3;2,5) + q(4;3,2)i.
Now, for operations like [q(3;2,1)+q(1;1,2)i] ⊕ [q(2;3,4)+q(2;0,1)i], the real structures are appended, and the imaginary structures are appended. This becomes q(3;2;2;3,1,4)+q(1;1;2;0,4,1).

Property 20 - Quaternions and up

Now for more complexity. We can have quanternions in both the planes and structures.
For example, q(2+4i+3j+11k;3+11i+6j) is a valid structure, along with q(2;4,3)+q(54;21,34)i+q(45+121i;34)j+q(451;37+29i,20i)k being a valid complex structure.

For the names of these structures, they all use the same naming scheme as discussed in several parts of the documents. For distinctions in each imaginary level, we add an additional numeral prefix at the start for each of the levels’ names.

The real structure has no additional prefixes.
The i level has: Duex-
The j level has: Truex-
The k level has: Tetrex-
The l level has: Pentex-
The m level has: Hextex-
The n level has: Heptex-
The o level has: Octex-
etc…

For the names of planes though, using the prefixes from above would become confusing. For this, we’ll use Greek numeral prefixes, and having distinctions for each imaginary level.
The real number level has: Proto- [but only if at least one imaginary level is present.]
The i level has: Deutro-
The j level has: Truto-
The k level has: Tetreto-
The l level has: Penteo-
The m level has: Hexteo-
The n level has: Hepteo-
The o level has: Octeo-
etc…

For structures like q(3;3+3i+3j+3k), the naming goes as: Tro-(Proto-Three, Deutro-Three, Truto-Three, Tetreto-Three). The names may not be the best to show this off, but it’s difficult to establish names for crazy numbers like this.
For more longer arrays like q(3+3i+3j+3k;4+4i+4j+4k,5+5i+5j+5k), it goes as: *cough cough…*
(Proto-Trero, Deutro-Trero, Truto-Trero, Tetreto-Trero)-(Proto-Four, Deutro-Four, Truto-Four, Tetreto-Four)-(Proto-Duquintune, Deutro-Duquintune, Truto-Duquintune, Tetreto-Duquintune). Phew, that is a mouthful… [pls give me ideas on simplification for this…]

Property 21 - Planes with numbers larger than eternity

If a plane contains a number larger than the number that triggers the overflow increment, in this case, eternity (□), that plane still resets to zero (0) and increments the next entry in the right by one (1), even if it clearly is way larger than eternity (□). It doesn’t automatically adjust the plane that was resetted to a number in the acceptable range.

Like, for example, ApG-DØS I-Mysterker (which is obviously larger than eternity) [used AP as an example]: q(0;AP) = q(0;0,1).

Because of the hierarchy, no amount of x_0 greater than eternity can surpass q(0;0,1).
I will clarify that this is ONLY if the number were to be used in this notation. ApG-DØS I- Mysterker does surpass q(0;0,1), but if it were to used in here it wouldn’t change the output.

But, numbers beyond eternity technically can output an array larger than q(0;0,1), but you have to *manually* map them to said array in order for this functionality to work. Same can be said about quantumnumbers that are quantumly larger than eternity.
Because this notation is so new, there is no current mapping of larger values to quantum notation arrays, but you can help out with that when the notation is complete. [This text in red will be removed when this is complete]

Property 22 - Quantum Googology’s Reset Theory

Because planes can have properties from normal numbers independent to each other and that the number properties are isolated to that plane only, it makes sense to question whether quantum googology’s reset theory works in here.

As a TL;DR, yes, the reset theory does work in here when a plane hits positive/negative point and return to zero, but ONLY in that plane can this be active.

For the long answer, each plane's number is in different "size" dimensions but they still combine together to form larger (or smaller) forms of numbers. These dimensions don't interconnect and thus, the planes have their properties isolated to their dimension only. They can only interact with each other when that plane overflows and needs to reset but stay large.

Since eternity (□) is larger than positive/negative point (+/-), where the reset theory applies to numbers, a plane that hits positive/negative point (+/-) can still go negative and go back to zero, and thus apply the reset theory. But again, what I established is that planes' properties can't affect other planes, so the reset theory only affects planes that have already gone through the journey to get back to zero.

This means that yes, bitsy quantumnumbers from other paths exist, like q(A[0];0,1). To make it easier to tell what planes already hit the reset theory threshold, we place a subscripted "R" on the left of the semi-colon or comma, or make a smaller "r" for this, like q(A[0]$_R$;0,1) and q(A[0]r;0,1), respectively.

The plane properties only affecting the plane they're in behavior is hardcoded, which makes it impossible to compare my quantumnumbers with other quantumnumbers that are reset theorized.

Also, if there are any future numbers that mess with the number logic, they are *all* isolated into this notations planes just like the rest and, unless it is a structural operation, it cannot affect all planes at once.

Update: This is true for the small scale of reset theory and Negativology's reset theory, but the quantum notation and all of it's variants are not beyond Absolute Aperdinal[sourced], since besides some Aperdinologisms bypassing this theory, the actual reset theory for quantum googology breaks all known numbers. There's much bigger numbers than the quantum notation, like Ultreta, which is the omega of this whole notation.

This still does not make it possible to compare my quantumnumbers to other quantumnumbers because of many other reasons already explained in this video. The reset threshold for incrementing planes are far too small for the real reset theory to even kick in, and this is before any kind of reset theory even started.