Beatrix matrix system sets number record!

by F. Gerard Lelieveld
mathemagical artist

A public gift of the largest natural number in mathematics, dedicated to the departing Queen Beatrix of the Netherlands, in gratitude for the prosperous period of her reign. Created for the occassion of the coronation on 30 April 2013 of her son King Willem Alexander, ruling with the lovely Máxima of Argentina on his side. ***„Viva la Reina!"***

A deep matrix called BTRIX is set up for the Big number championship against Chris Bird's BEAF algorithm. Subsequently we crown it with the mindboggling *"Beatrix Máxima"* with good hope the Dutch royal family will accept their new world record number, following the national festivities in the year 2013.

Big numbers are numbers without practical purpose, neither outside nor inside of mathematics. But we can define ever larger structures and smarter algorithms to create increasingly Big numbers.
By comparison this world is rather small, we cannot resolve those numbers completely.

Index

"Beatrix matrix system sets number record!" April-June 2013…
Mathemythical article by Giga Gerard on the construction of a Royal Dutch largest number.

- Structures by RepExp `F(..n..) :3: = F(F(F(n)))`
- Conventions `Btrix[a] = a`
- Choice rules `a,b, = a,b = b`
- Adding by abacus `a,b,1 = a,ab, = ab`

- **...to Be superstars**
  - Multiply over 3 entries `a,b,c = a*c+b`
  - Powers to Tetration `a,,,b,c = a***c1+**b`
  - Linear arrays `a,,{c}b = a*··b *:c`
- **Bea's dimensions**
  - Second row `a,b,[1]1c = a,{b}a,[1]c`
  - On the plane `a,b.,[1]..1 :c` **≈>** `EAF(a,b,b,c)`
  - Cubic and Dimensional `a,b,[c]1` **≈>** `EAF(a.,b..) :c1`
- **Beat nested arrays**
  - Entry in hyperspace `a,a,[a,b]1` **≈>** `EAF(a,b,[3]2)`
  - The hyper-row `a,b,[,{c}1]1` **≈>** `EAF(a,b,[c]2)`
  - Dimensional separators `a,a,[Dim]1` **≈** `EAF(a,a,[Row]2)`
  - Twice nested row `a,a,[,[Row]1]1` **≈** `EAF(a,a,[Dim]2)`
  - Nesting depth `a,b.,[..]1 :c:` **<≈** `EAF(a,b.,[1..]2.) :c:`
- **Btrix deep attachments**
  - Dual deep
  - Serial deeps

Start to count

A superstar? Well right you are!
– John Lennon (1st Beatle)

How do people define a largest number, when she can always count `1` more?

Only ones

Start with `1` (the unit *one*) which is next to nothing (*zero*) and then add up all natural numbers as a series `1...` of units one. Childlike *finger counting* is our final system of choice, specially for writing numbers that have become unwieldy large.
From input to output – all expressions in this article are (in theory) to be reduced to a string `1..` (of characters `1` repeated `n` times). So the final evaluation is a string (or word), and this word (as well as its size) is the venerable variable number value `n` itself.

```
n = 1.. :n
mn = 1..1.. 1:m 1:n = m+n
```

That `:n` right after the expression indicates the `1` is to be written `n` times.
Simple counting shows that a maximum number `m` is hard to establish. Indeed you can easily add some number `n` to topple the World Record through `mn`. A truly democratic pursuit!

In this article we will climb the most majestic mathematical heights together. Standing upon the

shoulders of gaint number enthusiasts, and the work they have done in the direction we want to go, which is… *"Big, Bigger, Biggest!"*

Big

The Big part is about the construction of natural numbers. The normal quantities people use are not that big. Fresh up on your old school arithmetic and you are all ready there.

Decimal numbers

The familiar decimal system defines small strings `{0,1,2,3,4,5,6,7,8,9,10,...}` in a unique number code that is humanly readable. This number system with its arabic numerals is not so easy to learn. (you've done it, congratulations!) For writing Big numbers its rules are not helpful at all, but for people with small needs and ten fingers it has its benefits.

Even science journalists tend to get confused and make mistakes when calculating with large numbers.
The quantities dealt with in science and finance, such as a million `1000000 = 1E6`, a billion `1,000,000,000 = 1E9` and a trillion `1E12` have many decimals, but are often estimates.

It is easier to count decimal zeros, than to write large numbers precisely.
For example: the public debt of the United States is about `2E13` U.S. dollar, a `2` with `13` zeros, as it grows multiplied by a percentage plus one, no one knows it exactly.

Scientific E

Scientific notation keeps track of the number of decimals. A scientist simply adds the number of zeros (power of ten) of two numbers to multiply them.

```
mEn = 10..*m 0:n
    = 10*..m :n = m*10^n
mEn*pEq = (m+p)*10^(n+q)
```

Some extreme examples of scientific and exponential notation:
There is a physical limit to the amount of information we can manage. In the present universe (`8E60` Planck moments after the Big Bang) the estimated number of quantum bits is `1E120`, therefore the binary state of our cosmos expresses at most a size `2^10^120` *power tower* (power of a power). And the history of this universe can be stored on a disk as a random number smaller than `1E1E180` (the laws of nature should act to compress this number).

Somewhere inside this small universal number, a subsequence of quantum bits that is fighting entropy expresses a very much larger number. Can this be the work of God?

They help us deal with some *"larger than life"* numbers, but decimal notation, its scientific extension by `E` and the elementary operators `+ * ^` are small sidesteps in our quest.
Sbiis' cascading E makes a jolly good attempt (we reviewed his hyper E notation) of extending the function of `E` to a workable system for Big numbers. But when we discuss some older number records we prefer the common extension of powers `^` to Knuth's arrows.

Boundary of arithmetic

We colour old school expressions brown, using the standard operator signs:

- `+` plus to add `a+b` (take `a`, add `1`, repeat `b` times),
- `*` star to multiply `a*b` (start with `0`, add `a`, repeat `b` times),
- `^` or `**` for powers `a^b` or exponentiation (init `1`, multiply by `a`, repeat `b` times),
- `^^` or `***` for tetration `a^^b` (init `1`, lift `a` to this power, repeat this `b` times).

Note that `10^n = 1En`
and let `10^10^n = 10^1En = 1E1En`.

The higher superpower operators just repeat the preceding operators.
For example, *tetration* `a***b` repeats `a**.. a:b` (a *power tower* of exponents `**a`).
Here, when `a=10` we have `10^^b = 1E..1 :b = E1#b` in hyper-E.

Higher up-arrows `^···` are reduced first (majority precedence), then come superstars `*···` lowest first (minority precedence). Equal operators are resolved from right to left.
Addition without an operator `ab` immediately takes effect. The plus `+` postpones addition (without brackets) until the other operations are done.

Instead of writing `c` arrowheads in `a^··b ^:c` we can count up-arrows `a↑{c}b` by RepExp (our simple RegExp for maths) or in a 3-entry array `a→b→c` like *chained arrows*.
John H. Conway's chained arrow notation harnesses a very powerful yet natural algorithm (a row of double recursions), that we shall relate to on occassion.

Arithmetic has no boundaries, as long as we know how to expand its rules. The principle of recursion theory is to count a class of functions and to operate on its index directly. Then expand the function array and eventually the rule system, so that we can recursively increase the previous index and explode the array structure.

Historical records

People used to take large numbers seriously, going to great lengths to find larger ones, I suppose. We examine the history of Big number records – from the ancient Greeks to a Cheltenham asperger with a degree in maths.

Archimedes *octads*

The history of Big numbers starts with a letter by Archimedes in the 3d century BC sent to King Gelon β of Syracuse. Popularly known as the Sand Reckoner, Archimedes' letter calculates the number of sand grains that "the sphere of the fixed stars" can contain. Archimedes gives an estimate of at most `10^63 = 1E63` sand grains, expressed with the octads (*myriad myriads* ordered in periods), which he had introduced in an earlier mathematical treatise.

Before the Greeks named `1E8` as a myriad myriads. Try to count a *googol* `1E1E2` with that! Archimedes would say a googol equals *"a myriad of octads of the 13th order"*. But the *googolplex* `1E1E100` lies out of reach for Octads.

The last number in the Octads system `10^(8*10^16) = 1E8E16` would have been the ancient Greek number record.

His system's expressive power leads Archimedes to the conclusion that the numbers constructible in mathematics are much larger than any quantity in the physics of the universe. And thus he puts his King's mind to rest.

Buddhist *unspeakable tower*

End 7th century AD the Chinese empress Wu commissioned buddhist monks with the translation of the Avatamsaka Sutra from Sanskrit. In this magnificent scripture the empress would have read a description of the largest quantity ever conceived by man.

The translation by Siksananda has the initial number of a hundred *laksha* `10^7` squared 103 times till the incalculable *asamkhyeya* `10^(7*2^103)` is observed (near the auspicious `2^2^108` and `10^10^32` power towers). Then by double squaring it reaches the *unspeakable* `10^(7*2^119)` and the *untold* `10^(7*2^121)` – numbers used in the following poem in the Avatamsaka (Thomas Cleary, "Flower Ornament Scripture", chapter 30, page 892).

> *The atoms* [containing untold lands] *to which these* [atomizable] *buddha-lands are reduced in an instant are unspeakable, and so* [atom to `10^(35*2^119)` atoms] *are the atoms of continuous reduction moment to moment, going on for untold eons...* [up to `10^10^2^124` atoms]
>
> *Counting this way* [power upon power] *for unspeakable eons, using unspeakable numbers, counting eons* [recursively] *by these atoms...*
>
> [Siksananda's total `10^^(10^(7*2^119))` atoms | lands | eons]

These verses express a number of about `10^..1 :2^2^124` by means of a power tower of *unspeakable* size. This record number from the literature of ancient India and China is the first example of *tetration* in the history of mathematics.

Guinness *Graham's number*

David Hilbert in *On the Infinite* (1925) proposed that a double recursion (his algorithm equals our superpower stars) increases faster than any *primitive recursive* function. This was proven by Wilhelm Ackermann in his 1928 article, and such functions (over the superpower index) are since called Ackermann functions.

In 1927 the Dutch mathematician Bartel van der Waerden founded the theory of arithmetical progressions in Ramsey Theory. Upper bounds for his numbers were once estimated using double recursion (but nowadays by a power `2^k#6` tower).
Ronald Graham is an expert of general Ramsey Theory, where the limits can be pushed a lot further. He came up with this huge number, to illustrate a powerful theory…

Martin Gardner's 1977 version of Graham's number once featured in the Guinness book of World Records as *"the highest number ever used in a mathematical proof"*. This record number counts `64` layers of up-arrows (double recursions), and is famously called *"Graham's number"*.

Graham's number is an upper bound for the `n`th dimension, where connecting points (vertices) in a 2-coloured hypercube forces a certain progression of colours (on the edges). The exact dimension where such a progression must exist is probably quite low, but while computers paint hypercubes at random up to exhaustion, a solution is still not found.

But the final number of Graham & Rothschild's original 1971 article has just 7 double recursions nested (of a special Ackermann function). Although their generalized formula seems capable of producing much larger upper bounds (for more exotic Ramsey problems than to solve a certain progression in a 2-coloured hypercube).

We will express both record numbers by nesting *:n:* chained arrows to their respective depth. Appending #n in a HYPER expression achieves the same iteration.

2→3→(..12..) *:7:* = 2→3→12#7 (original Graham's)

3→3→(..4..) *:64:* = 3→3→4#64 (famous Graham's)

It is awkward to accept the latter number as a record, because it resulted from a miscalculation (bake the blame). The earlier bound by the enigmatic Ron Graham is way too high already!
It was lowered by Saharon Shelah and now a proper upper bound (for the solution of a Hales-Jewett hypercube) can generally be expressed by some 2→k→5 class function.
Higher bounds for Ramsey problems are to be expected from certain sets of complete subhypergraphs. We question if any colour recipe for combined graphs requires Ackermann functions to express its upper bound, like Graham's hypercube problem used to do.

Bowers *meameamealokkapoowa oompa*

A clumsily defined number record from a Big number enthusiast from Tyler, Texas.
The *meameamealokkapoowa* numbers were coined by Jonathan Bowers at the end of his explosive list of names for Infinity Scrapers.
But we lose track on the way he gets there, at the Legion Legions structures – wild ideas that resulted from his email sessions with John Spencer. For us Bowers' Infinity Scrapers are still acceptable somewhere in the *Legions*, it's there the trail becomes misty.

High up in his number tree Hedrondude Bowers loses sight of his prime invention – the upload methods – when he places novel systems on top of the old. By nesting higher structures to the right, he severs the connection with the engine of his airplane – in effect making a fresh start from that point on.
Best is to let all structures increase (explode) from the prime entry below – by *upload* of its value to function as an index for the new structures. This should be well-defined – specially at the moment of the jump, the parachute should open ;-)

The third man involved in this record was Chris Bird, who on his own supplied a definition for the Meameamealokkapoowa Oompa, which requires a study of Bird's system to begin with.
Chris tells us his array notations easily topple all of Bowers' structures. We believe him, Chris Bird is the ruling world champion, the man of Big numbers to beat.

Bird *beyond nested arrays*

At present the best algorithm to construct the largest natural numbers, is the hyper-nested arrays of Christopher M. Bird from Cheltenham in England. Bird's arrays were originally inspired by the upload rule and hyper-dimensional structures of Jabe Bowers, who coined the term BEAF. But when we write BEAF you should think *"Bird's Extended Array Functions"* – Bird defined these algorithms and extended the array structures properly.

Greetings from The Hague, Holland! We are keen on designing a BTRIX matrix that improves on BIRD III – the best system of Chris Bird – his *Beyond Nested Arrays III*.
Our goal is to create a new system, that runs rule-significantly faster than the final function in BIRD III, the current world record holder `H`:

```
H(..3..)  :H(3):    (eternal laughter)
                = H(3#1#2 = H(3#H(3)
```

The former expression applies our REPEXP notation. Two words are repeated from both ends of the expression inwards up to the double dots `..` in tandem `:k:` times.
The latter expression is equal, but written with HYPER hashes `#` and single `(` brackets.

In this article we'll continue to measure our BTRIX matrix against Bird's system *on the fly*. If we can manage to jump from the nest and spread our wings, an exact record `Máxima` number can surely be constructed *high in the sky*.

Rayo *afterwords*

We feel the record number of the logician Agustín Rayo, *"the smallest number bigger than any finite number named by an expression in the language of first order set theory with a googol symbols or less",* apart from its fundamental failure to result in an exact number, is contrived in its motivation, and (surprise?) far from optimal.

The second order definition of Rayo's number by Gödel-alphabetization overlooks the true mechanism for creating Big numbers. Why?

Any expression can be translated to a simpler expression with less symbols and yet in the end be reduced to a larger number `1...` if only the rule system is Big enough. So the size and complexity of the Turing machine itself should be taken into account, not just the sizes of the tape it reads.
It is the typification of symbols itself that we seek to subject to recursion. *"Symbol"* is too fleeting a concept in the context of Big number algorithms.

Eventually we'd like to upload the size of the alphabets and the indices of Turing machines way back from below. We think interconnecting all these virtual machines leads to mind-bigglingly Bøg numbers. Rayo can keep his set-theoretic universes separate, but these stay mighty inefficient compared to running an ongoing recursion in between. Why?

Compare the upload system of Bowers & Bird with the archetypical *download systems* – the HYPER iterations of Sbiis always stay on top, and Conway's *chained arrows* never replace higher entries by previously incremented lower entries. Given the same alphabet size and symbolic complexity *downloads* result in Bigger numbers than *uploads*.
The down and dirty upload of BEAF, with its exploding higher array structures, gains speed (this is a conjecture to prove). It will one day upload new signs to alphabets and climb the hierarchies of hyperarithmetical Turing machines…

To *B* or not...

Explains the essence of number expressions `a,b` from scratch. People start to pay attention here!

Structures by RepExp

The highest principle for creating algorithms is ***frugality*** – to maximize the effect (function speed, Bigger numbers) with minimal means.

- Minimize the length of the expression string.
- Use less character types (e.g. evaluations in Btrix don't insert brackets).
- Restrict the rules to their bare necessities.

Let Btrix count down variables fully. Except for the initial entry `a` at the left, we allow a void (number `0`) in all positions. This rules out multiple mixed separators, keeping our algorithm simple.

The governing system of Btrix interprets all kinds of general recursive structures numerically. But she has a character rewriter (our Abacus machine) under the hood. Our matrix function, superficially dealing just with numbers, is living in a programmer's world.
Ideally we should always point out the way a new jump to a higher structure is represented by primitive rewrite steps inside a word (string).

From traditional Regular Expressions (RegExp) we may use the quantifiers `?` or `{0,1}` for none or one occurrence, in context `{k>0}` or `{1,}` for one or more, and `{k≥0}` or `{0,}` for any number. In general `{k}` for exactly `k` copies, and `{m,n}` for at least `m` and at most `n` copies of a preceding word: a single sign `x` or `(X)` bracketed string.

Our new ***RepExp*** meta-notation to select and repeat parts of an expression in rules, is quite practical and concise. It applies a mathematical dot notation *`:k:`* to word repetition, and usually regular expressions to repeat `{k}` single characters. Words can run from either side or from single dots `.` placed at the beginning of the *:left* or *right:* repeated words.
Show a few examples of RepExp.

`x.. :3 = x.. x:3 = x{3} = xxx`

`F(..m..) :3: = F(F(F(m)))`

`A.n1,..Z :2 = A(n1,){2}Z = An1,n1,Z`

`aaaa..b..b.c`$_i$`.. aa:0 2: :4 = aabbbc`$_0$`c`$_1$`c`$_2$`c`$_3$

`A.L`$_i$`..,{5}..R`$_j$`.Z :3: = AL`$_0$`L`$_1$`L`$_2$`,,,,,R`$_2$`R`$_1$`R`$_0$`Z`

The word to Rep(eat) from left to right in the Exp(ression) runs up to the double dot marks `..` and we can connect this to a right word repeated inwards (to the left) by a second pair/trio of dots. The quantifier in this tandem repetition *`:k:`* counts how many word copies are placed on each side.

Conventions

Our variables are ***greedy*** for number units, naturally for the sign `1`. So when a variable `n` sits next to a wildcard `X`, the proper substitute of `X` won't have a number expressed on that end.

`n = 1.. :n≥0 = 1{0,}` (RegExp unlimited)

`Xn ≠ Y1n` (greedy variable `n`)

`_n_ ≠ _n1Y` (underscore wildcard for passive parts)

We use the = sign when two expressions are equal, and when the right (decoded - output) expression follows from the left (encoded - input) in the evaluation train to natural number.
Expressions are *not equal* ≠ when one side cannot be reduced to the other.
We use the ≡ sign for the ***equivalence*** of expressions in different formats, or to justify a substitution (exchange) of substrings within an expression.

Our BTRIX matrix system applies two conventions: one at the beginning and one at the end of time (of the evaluation of an expression to Big number).

`0.0` `BTRIX[X] ≡ X` (simplify)

`0.1` `BTRIX[a] ≡ a = a` (identify)

The ***initial convention*** is to write the matrix in a simple format – without function title and brackets – which is sufficient because the evaluation rules of BTRIX won't nest an expression in place of a variable (so common in other systems).
The ***final convention*** is the identity function that reads the number result out. It's a string of `1..` (from here on your dedicated ABACUS could translate it to the preferred format ;-)

Because of our initial convention we feel fine that `BTRIX[a]` is defined now. But because our first rule, the *rule of choice*, chooses to discard `a` we might as well have left it undefined.

Choice rules

Our system starts with the simplest of rules. Addition happens of itself when placing two variables `ab` holding number units `1..` next to each other. There's no use adding a rule to arrange for that, as this could only mess things up.
It is when you want to keep *here* what you are putting *there* (and try to repeat that), that complexity arises. A life lesson.

Anyway, more fundamental than any arithmetical operation is *selection*, the operation of choice. This rule performs the last act of BTRIX to evaluate an expression to a (natural) number.
First the initial case where `b=0` reduces to `0` and then the standard *rule of choice*. This rule is extended for when trailing commas have been kept, and must now be discarded.

`1.0` `BTRIX[a,] ≡ a, =` (`a` void)

`1.1` `a,b = b` (choice `b`)

`1.2` `a,b,{k} = b` (choice, ignore trailing commas)

To drop or keep trailing structures is a convention, with no significant impact on the resulting Big numbers. This was a discovery by Jonathan Bowers, an insight he had in array structures.
Bowers wanted to count off the pilot iterator and just leave the residual `,1` be, even if it is never uploaded again. Rule **`1.2`** can be extended for this purpose, ignoring the tail separator structures until the very last evaluation, when no iterator entry remains in `$` on the right.
In contrast Bird uses a comparison system to determine whether left separators `,[L]1,[R]` are smaller than right, and therefore are trailing and consequently dropped off.

```
2.1    a,Y, = a,Y    (drop comma tail)
```

Rule `2.` is optional. If you work it out there's no need to remove trailing separators. You can either let them be until the very end, reuse some of them, or optionally clean up during evaluation.

Adding by abacus

To add two numbers `ab` is pretty straightforward. But how do you copy a number to add it, when you can only distribute a few characters at once? Inside `4+2` for example?

```
1111,11,1 = ,1111,1,11,    (prefix , and insert 1, to split base)
          = ,111,11,111, == ,1,1111,11111,    (doubling)
          = 1111,111111,    (carry 1 over and drop prefix)
```

Prefix *null* `,` parameters for special machine modes!
All `colours` are there to help your human eyes.
Next example `3+0` of adding to zero, coded (with `,` for `0`) in binary.

```
111,,1 = ,111,1,,
       = ,11,11,1, = ,1,111,11,
       = 111,111,
```

Scheme for adding by doubling in ABACUS, with wildcards `W`$\geq\varnothing$, where `a>0` is natural.

```
a,b,1X  =  = ,a,1,b,X   (lower anchor)
,m1,1n,Y  =  ,m,1n1,1Y    (double up)   == ,1,a,ab-,X
,1,a,Z  =  a,1Z   (hail the anchor)   = a,ab,X
```

Word repetition is our main tool in the definition of rules. Here we add numbers on a primitive level – doubling by copy `1` and substitute `1,1` or `,1` in second place.
Check the initial case.

```
1,,1 = ,1,1,,    (put 1+0 in doubling mode)
     = 1,1,    (finish = 1 in matrix mode)
     = 1,1    (drop trailing separator)   = 1    (choice)
```

Only a pre-mathematical machine writes *null mode* `,` prefixes. We have to do without the natural void (try unit `0'` instead) in the first entry. At the *anchor* position `a≠0` is mandatory.
The ABACUS machine abides, with plenty of shekels to spend in the red. It can resort to lengthy prefixes `,{u}W`$_i$`..,{u1}a,Z` *`:v`* where in `W` all `,{t<u}` to boil our rewrite rules down to the simplest of commands. [it is so vicious :!]

...to *Be* superstars

People will be thrilled to see BTRIX on front row, running a tight race against superpower stars!

Multiply over 3 entries

The *motor rule* of BTRIX increases the value `b` in the *basket* by adding `a`. This is truly fundamental, but significantly slower than the substitution rules of other Big number functions.
The competition speedily nests predecessor versions of the expression as it grows, while she just piles up *apples* `a..` forever and ever. What a drag queen!

Follows the motor rule **`3.`** of addition, which amounts to multiplication by repetition == of steps.
Afterwards the rule of choice **`1.`** is applied to free the result.

`3.0` `a,b,1c = a,ab,c` (motor)

`a,b,c == a,.a..b, :c`
`= a..b :c ≡ a*c+b`

For examples, fill in the above formula, where (given `a` and `b`) appending `,1` adds `a` to `b`.
To repeatedly add `a` (initially to `0`) amounts to `*` multiplication by `,,` (double comma).

`a+0 ≡ a,,1` ***3.*** `= a,a,` ***1.*** `= a`

`a,b,1` ***3.*** `= a,ab,0` ***2.*** `= a,ab` ***1.*** `= ab ≡ a+b`

`a,,c =` ***3.*** `= a,a{c},` ***2·1.*** `= a{c} ≡ a*c`

At this stage the only variable shift is by `a` up one position to `b`. There is still no upload higher up, which gains power the further it travels. The first such *upload rule* – moving the multiplication gathered here in *basket* `b` one entry up – is necessary for exponentiation.

Eventually over the full row BTRIX will reach superpowers (double recursion). We translate this concisely to Bowers' original BEAF system or to its stand-in EAF over just `3` entries.
Start by expressing addition with `EAF(a,b,1) = EAF(a,b)` then multiply by `EAF(a,b,2)` and powers at `EAF(a,b,3)` which is where Bird's `BEAF(a,b,1)` starts.

`EAF(a,b1,c2) = EAF(a,(a,b,c2),c1)`
`== EAF(a,..1..,c1) :b1:`
`= a*{c}..a :b = a*{c1}b1`
`= a^··(b+1) ^:c = BEAF(a,b1,c)`

The BTRIX row stores more variety (has a better resolution) than BEAF style notation, but numbers expressed by linear arrays in BEAF are that much Bigger.
At the linear section below the first row of EAF and BEAF is defined for comparison.

Powers to Tetration

Over four entries BTRIX `a,,,d` lifts a number `a` to the power `d`. For this we introduce two new rules to upload variables from left to right.

You can only put an *apple* `a` in the *basket* `b` when you count a higher entry down. In rule **`4.`** we meet that iterator entry at `d` – one entry further than usual in rule **`3.`**
Rule **`5.`** loads a non-zero value from the basket up to the empty entry at `c` here (or generally to the rightmost entry of the empty series starting at position `c`).

`3.1` `a,b,1c,d {b≥0} = a,ab,c,d` (add `a`)

`4.0` `a,,,1d = a,a,,d` (put `a`)

`5.0` `a,b1,,1d = a,,b1,d` (upload `b`)

`= a,a,b,d` ***`4.`***

General rules – shifting values over the length of a row – are defined in the next section. We have to wait for the multidimensional BTRIX matrix to approach `4` entries of BEAF.

We will translate ≡ expressions of linear BTRIX to a *postponed operator* format, where the operator `+*` is a *postponed multiplication* in `x**y1+*z`, reduced to `x**y+*x*z` and further to `x*..z :y1` with the factor `z` put on top of a series of factors `x`.

```
a,b,c,d1  = a,v,,d1  with v ≡ (a,b,c) = a*c+b      (3.)
          = a,,v,d   but if v=0                    (5.)
          = a,a*v,,d                               (4.)
          = a,.a*..a*v,, :d                        (3.)
          = a*..v :d1   = a**d1+*v                 (5·3.)
          = a**d1+*a*c+b                           (2·1.)
  or    = a**d+*a*0+a = a**d1  if b=0 and c=0
```

Here `a,,,b` is exactly equal to `a**b` exponentiation.
Similarly over five entries `a,,,,b` equals `a***b` tetration in BTRIX.

```
a,b,,,e1 = a,,,b,e
         = a,(a,,,b),,,e
         = a,a**b,,,e     (result from powers)
         == a,.a**..b,,, :e1
         = a***e1+**b  {b>0}

a,,,,e1 = a,a,,,e
        = a***e+**a
        = a***e1

a,b,c,d,e = a,(a,b,c,d),,,e
          = a,a**d+*a*c+b,,,e
          = a***e+**a**d+*a*c+b
```

Apparently commas `a,{c}b` can function like countable operators for superpowers. This interpretation of functional separators in BTRIX is considerably slower than EAF `(a,b,c)` which expresses superpowers with two single commas.

Consider an even simpler system, where all separators are void and collapse to zero, so that ground level entries `n` are added (by default). Because the algorithm is undefined (disfunctional), such a *system normalizer* NIX can be used to classify levels of separators, provided we fill all structures with previous substructures (in the style of Bird's bracketed arrays), all of size `n`.

For starters `n,n = nn` and powers `n,n,n,n = n*4` and first row `n,.. :n = n**2` by rule of NIX, which drops all structure (on the left side) and adds up entries `n` to a class level norm `Ñ`, for now expressed with minority stars (on the right).
Our classification has a natural base `n` and will soon run on a par with Bird's `ω`-based separator spaces, which take off properly at `[n]` multiple dimensions. Bird attributes space `1` to the linear array, because he separates its substructures (normal entries) with `[1]` commas.

Linear arrays

The first row of entries in the BTRIX matrix results in the same function proposed by David Hilbert in his *On the Infinite* (1925, in the guise of a double functional algorithm).
The BTRIX separator commas `,{k2}` are comparably equal to *superpower stars* `*{k1}` and to Knuth's arrows `^{k}` (two “classes” apart in primitive recursion).

Despite that values are only being added, shifted and counted down, linear BTRIX covers the whole hierarchy of primitive recursive functions. She succeeds on the strength of her *upload rule*, which is a simpler rule than in Bowers' BEAF, but not significantly slower in principle.

3.2 `a,b,1Z = a,ab,Z` (motor)

4.1 `a,,{k1}1Z = a,a,{k1}Z` (put)

5.1 `a,b1,{k1},1Z = a,,{k1}1b,Z` (upload)

4. `= a,a,{k1}b,Z`

`== a,a,.a-,..b,Z :k`

Rule **3.** and rule **4.** both add *apple* `a` to *basket* `b`, after decrementing the first available number on the right. Upload rule **5.** counts off the active iterator (Bowers' array *pilot*) on the far right, then shifts value `b` to stop short and fill the empty entry before (Bowers' *co-pilot*).
Only at the start of an evaluation rule **4.** serves its purpose, later rule **4.** and **5.** can be applied in one go, culminating in a complete substitution over the free part of the array.

A peculiar consequence of having two upload shifts is that `a,1,,Z = a,,,Z {1∈Z}` and for postponed superstars `X+*{k1}1 = X` whilst `X ≠ X+*{k1}0` on the right.
We translate a row of entries in BTRIX to superpowers, achieving equivalence by postponed operators. Superpowers can be expressed with only `2` iterator entries in an *array function*.

```
a,b,{r1}1q = a,{r1}b,q
           = a,(a,{r1}b),{r1}q
           = a,a*{r}b,{r1}q   (superpower as before)
          == a,.a*{r}..b :q1
           = a*{r1}q1+*{r}b
a,,{r1}1q = a,a,{r1}q
```

```
            = a*{r1}q+*{r}a {q>0}
            = a*{r1}q1 {q≥0}
            = EAF(a,q1,r2) = BEAF(a,q1,r)
a.,pi..,q :r1 = a,v,{r1}q  where v = a.,pi.. :r1
              = a*{r1}q+*{r}v  == etc.
            ~> EAF(a,q1,r2) = a*{r1}q1
```

The previous section featured the "class 4" superpowers of *tetration*.
For an example of the evaluation train of `11,,,,,111` *pentation* (`2^^^3` or `2****3`) we refer to the table in the appendix.
There you also find the rules of *postponed superstars*. With this support system the function array of BTRIX can be translated precisely to operator format.

To use these systems for comparisons in the next chapter, we define the first row of EAF and BEAF – with rule precedence by list order – covering Bird's linear arrays. Here are no `0` entries.

```
 I.  EAF(a,b) = ab = a+b   (primitive)
       BEAF(a,b) = a^b = a*..1 :b
ii.  EAF(a,X,1) = EAF(a,X)   (finalize)
       BEAF(X,1) = BEAF(X)
III.  EAF(a,1,Z) = a = BEAF(a,1,Z)   (inner drop for rule 5)
 iv.  EAF(a,b,1,2Z) = EAF(a,a,b,1Z)   (upload rules)   &
      EAF(a,b.,1..,2Z) :k1≥2 = EAF(a,a,.1,..b1,1Z) :k
      so  EAF(a,b.,1..,2Z) :k≥1 = EAF(.a,..b,1Z) :k1
        BEAF(a,b1.,1..,1Z) :k≥0 = BEAF(.a,..f,Z) :k1
         (Bowers' motor upload)      f = BEAF(a,b.,1..,1Z) :k
 V.  EAF(a,b1,2Z) = EAF(a,(a,b,2Z),1Z)   (motor rule)
                  == EAF(a,..a..,1Z) :b:
       BEAF(a,b1,1Z) = BEAF(a,f,Z)   (initial case k=0 of rule 4)
                 &  f = BEAF(a,b,1Z)
```

The *Easy Array Function* (EAF) uploads entry `b` straight, instead of a predecessor expression. Now BEAF's results need not be greater than EAF's (due to this easy upload rule). We can just increase entry `c` to cause one extra nesting in `b` accordingly (let the motor rule do the work).

```
EAF(a,3,2,2) = (a,(a,a,1,2),1,2) = EAF(a,a,(a,a,a))
<~ BEAF(a,3,1,2) = (a,a,(a,2,1,2)) = BEAF(a,a,(a,a,a))
```

Observe that `EAF(a,b,1Z)` < `BEAF(a,b,Z)` ≤ `EAF(a,b,2Z)` falls within (an ever tighter) range (towards the lower EAF bound).

Bea's dimensions

People should appreciate the growth rate of these functions. Linear BEAF outruns BTRIX but we compensate for this, by means of an extra dimension counter.

Second row

Construction of a multidimensional BTRIX matrix in a few examples. Comparison with Bird's linear array of BEAF and its simplification to EAF (our *Easy Array Function*). Both systems were defined by a rule list for the linear array in the previous chapter.

Array rows in BTRIX are separated by `,[1]` commas. Dimensions in general are constructed by placing indexed separators `,[s]` between subordinate dimensions `s` inside the matrix. Bowers starts with the same dimension index `[1]` between rows, but Bird assigns `[1]≡ ,` to the single comma. We write Bird's row separators as `,[2]` in EAF essentials.

Choose to clean up trailing `a,b,{k},[1]Z` left commas, or keep and add them `,{kb}` to the next expansion (as Bowers suggested). Then `b` will obviously be much larger than `k`, so the resulting Big numbers (comma *drop* versus *keep*) won't differ so much.

```
a,b1,[1]1 = a,a,{b1}1 = a,{b1}a
          = a*{b}a {b>0} = a^{b}2
          = EAF(a,a,b1) = EAF(a,b1,1,2) = BEAF(a,2,b)
a,b,[1]2 = a,a,{b}1,[1]1 == a,(a,{b}a),[1]1
         = EAF(a,a,(a,a,b)) = EAF(a,(a,b,1,2),1,2)
         ≈ EAF(b,3,2,2) {a≈b} ~> BEAF(b,2,1,2)
a,b,[1]1c = a,{b}a,[1]c == a,(a,{b}a),[1]c
         == EAF(a,..b..,1,2) :c1:
          ≈ EAF(b,c2,2,2) {a≈b} ~> BEAF(b,c1,1,2)
```

The new rule **6.** of system BTRIX applied above on the 1st entry in her 2nd row to expand the 1st row. It assumes trailing commas `,,[1] ≡ ,[1]` are mopped up. We also could have kept trailing dimension separators for reuse, as argued above the difference is not significant.
Extended rules for *upload* etc. are listed in the coming section on higher (multiple) dimensions.

```
6.0   a,b,[1]1Z = a,a,{b}1,[1]Z
                = a,{b}a,,[1]Z = a,{b}a,[1]Z
               == a,(a,{b}a),[1]Z
```

Further construction with comparisons. The next row of BTRIX is appended, which in total adds just `1` to the 4th entry in EAF (and similarly in BEAF). All variables must be put `{a≥3}` in context.

```
a,b1,[1],1 = a,,[1]b1, = a,a,[1]b
           = EAF(a,b1,2,2)
a,b1,[1],2 = a,a,[1]b,1 == a,(a,a,[1]b),[1],1
           = EAF(a,(a,b1,2,2),2,2) ≈ EAF(b1,3,3,2) {a≈b1}
a,b1,[1],1d = a,a,[1]b,d == a,(a,a,[1]b),[1],d
```

```
            = EAF(a,..b1..,2,2) :d1:
            ≈ EAF(b1,d2,3,2) {a≈b1} ~> BEAF(b,d2,2,2)
a,b1,[1],,1 = a,a,[1],b, = EAF(a,b1,3,2)
a,b1,[1],,2 = a,a,[1],b,1 == a,(a,a,[1],b),[1],,1
            = EAF(a,(a,b1,3,2),3,2) ≈ EAF(b1,3,4,2) {a≈b1}
a,b1,[1],,1e = a,a,[1],b,e == a,(a,a,[1],b),[1],,e
             = EAF(a,..b1..,3,2) :e1:
             ≈ EAF(b1,e2,4,2) {a≈b1} ~> BEAF(b,e2,3,2)
a,b1,[1],{r1}1q = a,a,[1],{r}b,q
            == a,(a,a,[1],{r}b),[1],{r1}q
             = EAF(a,..b1..,r2,2) :q1:
             ≈ EAF(b1,q2,r3,2) {a≈b1} ~> BEAF(b,q2,r2,2)
```

Next rows in BTRIX will have a seemingly negligible impact compared to EAF with its fast motor rule which nests predecessor expressions. It is only slowly our queen matrix creeps near…

On the plane

We adapt the row upsize rule **6.** for the plane matrix (over multiple rows) of BTRIX. Supply the upload rules for entries over multiple dimensions in the next section.

On the plane the rightmost empty row (of a `,[1]..` series) has its size expanded (by `b`) and only its top entry gets filled (by `a`). Repeat rule **4.** and **5.** in tandem to fill the rest of the top row (with entries `a-`) and then rule **6.** will expand earlier rows (to sizes `a`).
Considering the size of the top structure is dominant, Bird's *angle brackets* algorithm to explode multidimensional arrays is not significantly faster than ours. After the top entry is counted down, lower structures have been uploaded from much Bigger baskets `b'` many times.
So the upsize (by `b`) of an expression `a,b,[X]1Z` in BTRIX matches the upsize (by `b`) of `a,b[X]1Z` in BEAF, even after all Bird's angle brackets `a<X'>b` are worked out!

```
6.1   a,b1.,[1]..1Z :k1 = a,a.,[1]..,{b1}1,[1]Z :k
                    5.= a,.,[1]..,{b}a,,[1]Z :k
                  8·4.= a,a.,[1]..,{b}a-,[1]Z :k
      == a,f.,[1]..Z :k1 & f = a,a.,[1]..,{b}a- :k
```

Our minus unit `-` subtracts `1`. Use the sign **≈** when two expressions are approximately equal, use **~** if an insignificant variable is lost. In a comparison `X` **~>** `Y` the number `X` is *next larger* than `Y` (so `X` comes after in Big succession).
From the royal `3` row matrix onwards we only compare BTRIX and EAF approximately.

```
a,b1,[1],[1]1 = a,a,[1],{b}a- = EAF(a,a,b2,2)
a,b1,[1],[1]2 = a,(a,a,[1],{b}a-),[1],[1]1
    ≈> EAF(a,a,(a,a,b2,2),2) ~> EAF(b1,3,2,3)
a,b1,[1],[1]1c = a,(a,a,[1],{b}a-),[1],[1]c
```

```
    ≈> EAF(a,a,..b2..,2) :c1: ~> EAF(b1,c2,2,3)
a,b1,[1],[1],2 = a,(a,a,[1],[1]b),[1],[1],1
    ≈> EAF(a,(a2,b1,2,3),2,3) ~> EAF(b,3,3,3)
a,b1,[1],[1],1d = a,a,[1],[1]b,d ~> EAF(b,d2,3,3)
a,b1,[1],[1],{r1}1q = a,a,[1],[1],{r}b,q
    ~> EAF(b,q2,r3,3) ~> BEAF(b,q2,r2,3)
```

Continuing this comparison over the BTRIX matrix, where linear size (the number of entries of her top row) runs on a par with the `3d` entry in EAF. Appending a next row on (the top plane of) her `2`nd dimension is like incrementing the `4`th entry of EAF (or BEAF) by `1`.

```
a,b,[1],[1],[1]1 ~> EAF(a,a,b1,3) ~> EAF(b,2,2,4)
a,b,[1],[1],[1]1c == a,(a,a,[1],[1],{b-}a-),[1],[1],[1]c
     ≈ EAF(a,a,..b..1,3) :c1: ~> EAF(b,c2,2,4)
a,b.,[1]..1 :e ~> EAF(a,a,b1,e) ~> EAF(b,2,2,e1)
a,b.,[1]..,{d}c :e ~> EAF(b,c1,d2,e1)
```

We've nested subexpressions in BTRIX basket `b` in exchange for decrementing her right parameter. Now we like to take a broader view. The number of rows `e1` of BTRIX plane is reflected in the `4th` entry in EAF, and the number of entries `d1` on BTRIX top row reflects the `3d` entry of EAF. We expect a similar reflection between both systems as we move to higher dimensions.

In the next section the concept of multiple dimensions will follow from linear, plane and cubic dimensions. All such structures get a class norm `Ñ` via system NIX, explained at ~0 above.

```
n,..,[1].. :n :n = n**3   (plane)
n**3,[2].. :n = n**4   (cubic)
n**n,[n-].. :n = n**n1   (dimensional)
```

We can classify the next higher separator `,[,1]` in BTRIX (or equivalently `[1,2]` in Bird's BEAF) with norm `Ñ = n***2` (where Bird jumps to level ω of separator spaces).

Cubic and Dimensional

Watch the number of planes in cubic BTRIX match the `5th` entry in EAF.

```
a,b,[2]1 = a,a.,[1]..1 :b = a,a.,[1]..,..1 :b- :a
    ≈> EAF(a,2,2,b1) ≈ EAF(b,2,2,1,2) {a≈b}
a,b,[2]2 = a,(a,b,[2]1),[2]1 ~ EAF(b,3,2,1,2)
a,b,[2]c ~ EAF(b,c1,2,1,2)
a,b,[2],1 ≈> EAF(a,b,2,1,2) ~ EAF(b,2,3,1,2)
a,b,[2],c ~ EAF(b,c1,3,1,2)
```

```
a,b,[2],[1]1 = a,a,[2],{b}1 = a,a,[2],{b-}a-
     ≈>  EAF(a,a,b1,1,2)  ~>  EAF(b,2,2,2,2)
a,b,[2],[1]c  ~>  EAF(b,c1,2,2,2)
a,b,[2],[2]c  ~>  EAF(b,c1,2,1,3)
a,b.,[2]..,[1]..,..c :f :e :d  ~>  EAF(b,c1,d2,e1,f1)
a,b,[3]1 = a,a.,[2]..1 :b  ≈>  EAF(a,2,2,1,b1)
```

We extend the rules of BTRIX to expressions in a multidimensional matrix.
Variables have values `v>0` as determined by context (sound use & rule order).
Rule **5.** can be presented as a combined rule to *upload* `b` and *put* `a` in place.
Rule **6.** expands previous dimension sizes, with `,[]` a regular comma between entries.

1.3 `a,b.,[si]..  :k  = b`

2.2 `a,Y,[s]  = a,Y`

3.2 `a,b,1Z = a,ab,Z`

4.2 `a,.,[si]..1Z  :k1  = a,a.,[si]..Z  :k1`

5.2 `a,b1.,[si]..,1Z  :k1  = a,.,[si]..b1,Z  :k1`
4. `= a,a.,[si]..b,Z  :k1`

6.2 `a,b1$,[t1]1Z  where  $ ≡ ,[si]..  :k  by  8.  {sk≥t1}`
`= a,a$.,[t]..1,[t1]Z  :b1`
6·8. `= a,a$.,[t]..,[t-]..1,[t1]Z  :b :a` etc.

7.1 `,[] ≡ ,` (make `,[-] ≡ 1` sense?)

8.0 `,[s<t],[t] ≡ ,[t]`

The optional rules **2.** and **8.** discard of trailing outer and inner separators.
Bird clears away smaller left separators to clarify the use of other rules, but Bowers suggests we can keep them. We can choose either strategy – for the Bigness of numbers to either keep or drop these superfluous commas is mostly insignificant.

Every next dimension of BTRIX is equivalent to an extra entry in the linear array of EAF (and likewise Bird's BEAF), until our multiple Dim and his first Row meet.

```
a,b,[4]1 = a,a.,[3]..1 :b  ≈>  EAF(a,2,2,1,1,b1)
a,b,[m1]1 = a,a.,[m]..1 :b  ≈>  EAF(a,a.,1..,b1) :m
```

Our BTRIX matrix barely catches up with Bird's linear array, whose size equals her separator index `,[m]` for multiple dimensions. However, this is no big deal in the hyperdimensional realm we are about to enter.

Rules specific for multidimensional arrays in EAF (to rival BTRIX hyper-row in next chapter).
Count down entries to `1` (nowhere `0` void). Apply rule precedence by list order.

Rule `4'` can be seen as an initial case of rule `6` with the same second entry `b` if we put `t=0` to rely on EAF's trivial separator cases as follows.

```
      let  ,[1] ≡ ,   &   ,[] ≡ 0
II.   EAF(a,X,[S]1) = EAF(a,X)    (drop trailing end)
      use  $ ≡ ,[s_i]1..  :r  {r≥0 s>0}
iv.   EAF(a,b,1,2Z) = EAF(a,a,b,1Z)  &'  {r>1 or s_0>1}
      EAF(a,b$,2Z) = EAF(a,a$b,1Z)   (upload 1b)
vi.   EAF(a,b1$,[1t]2Z) {t>0} =
          EAF(a,a$.,[t]1..1,[1t]1Z)  :b   (upsize)
```

The growth rate of our EAF system differs not significantly from Bird's multidimensional arrays. Below is explained why.
For example against `BEAF(a,2[2]1[2]2) = BEAF(a,a[2]a,a)` the next larger **<~** is about `EAF(a,3,[2]1,[2]2) = EAF(a,a,[2]1,1,2,) = EAF(a,a,[2]a1,a)` and so forth.

EAF adopts Bowers plan to leave trailing separators in between `,[S]1,[T]` if `{S<T}` and so we don't need Bird's complex subsystem for comparing separator arrays at all. Any extra structure that was left behind is eventually reused, but only after its parent space has been upsized anew by much bigger `b` than before. This dwarfs the extra upload of older remnants in EAF time after time.

Convince yourself the basket `b` grows bigger before each particular upsize. When the right side is `1` smaller since last time, and the middle part is counted down fully, the bigness of the former expression is transferred to the left side. The right stuff must then be concentrated in the number `b` (and a small part in trailing middle structures, until these are dropped in the end).

Our view of Bowers & Bird's <bracketed arrays> is that these explosions don't contribute much to the greater good. The upsize `b` of the highest free structure is most significant. In EAF all other free structures are upsized by `a` and in BEAF by `b`, but such secondary differences don't matter.
For suppose our upsize is by `b1` instead, then the direct penultimate upsize is by `a`, but once the highest structure is counted off (effectively) to size `b`, the subsequent upsize of EAF's penultimate structure will be unimaginably bigger (though still insignificant) compared to that of BEAF from `b`.

Taking the upload of a predecessor expression in BEAF versus value `b` in EAF into account, we can wrap up all their algorithmic differences as insignificant. Just write `EAF(a,b,2,[S]Z)` to compete against `BEAF(a,b[S]Z)` and the former expression is destined to beat the latter.

Beat nested arrays

People would weep, should BTRIX become too slow to catch up with Bird's algorithm. So they will leap for joy, when our matrix engages his BEAF in deep nested hyperspace.

Entry in hyperspace

With the next nested entry `,[,1]` in BTRIX, we take our first steps on the extra-dimensional level, also known as *hyperspace*. The numbers expressed here are comparable to EAF wrapping up his row of last chapter with its `,[2]` separator.

On the left BTRIX resolves multidimensional subexpressions to increment right entries, append rows, planes and multidimensional blocks, in order to get at the nested entries in her separator array.

```
a,b1,[,1]1 = a,a,[b]1 = a,a.,[b-]..1 :a
    ≈> EAF(.a,..a) :b1 = EAF(a,b2,[2]2)
a,b1,[,1]2 = a,a,[b]2 == a,(a,a,[b]1),[b]1
    ≈> EAF(a,(a,b2,[2]2),[2]2) ~> EAF(b1,3,2,[2]2)
a,b1,[,1]c ~> EAF(b1,c1,2,[2]2)
a,b1,[,1],1 = a,a,[,1]b ~> EAF(b,2,3,[2]2)
a,b,[,1],2 = a,(a,b,[,1],1),[,1],1 ~ EAF(b,3,3,[2]2)
a,b,[,1],,1 = a,a,[,1],b- ~ EAF(b,2,4,[2]2)
a,b,[,1],[1]1 ≈ EAF(a,2,b2,[2]2) ~ EAF(b1,2,2,2,[2]2)
a,b,[,1],[1],[1]1 ~> EAF(b,2,2,3,[2]2)
a,b,[,1],[2]1 ~> EAF(b,2,2,1,2,[2]2)
a,b,[,1] Dim ~ EAF(Row,[2]2)
```

A lot Dim of dimensions fly over the BTRIX bridge to add 1 yota to the plane of EAF.
The big question is whether BTRIX stays (on account of motor 3. which just adds) structurally slower than EAF. If she has to nest a hyper-array to approach the next linear array of EAF, she might never catch up. But now we will show she is not that bad.

```
a,b,[,1],[,1]c ~> EAF(b,c1,2,[2]3)
a,b,[1,1]1 = a,a.,[,1]..1 :b ≈> EAF(a,a,[2]b1)
a,b,[1,1]c ~> EAF(b,c1,2,[2]1,2)
a,b,[1,1] Dim ~ EAF(Row,[2]1,2)
a,b,[1,1],[,1]c ~> EAF(b,c1,2,[2]2,2)
a,b,[1,1],[1,1]c ~> EAF(b,c1,2,[2]1,3)
a,b,[2,1]c ~> EAF(b,c1,2,[2]1,1,2)
a,b,[2,1],[2,1]c ~> EAF(b,c1,2,[2]1,1,3)
a,b.,[e,1]..c :d ~> EAF(b,c1,2,[2].1,..,d1) :e
```

That first hyper-entry e of BTRIX approaches the second row of EAF in size. But it is in the same place as the multidimensional index, that matched the first Row of EAF. So the 2nd hyper-entry in BTRIX (which counts iterations over the 1st) is equivalent to the number of rows on EAF's plane.

```
a,b1,[,2]1 = a,a,[b,1]1 ≈> EAF(a,b,[2]1,[2]2)
a,b,[,2]c ~> EAF(b,c1,2,[2]1,[2]2)
a,b,[,2],[,2]1 ≈> EAF(a,b,[2]1,[2]3)
a,b,[1,2]c ~> EAF(b,c1,2,[2]1,[2]1,2)
a,b,[1,2],[1,2]c ~> EAF(b,c1,2,[2]1,[2]1,3)
```

`a,b,[2,2]c` **~>** EAF`(b,c1,2,[2]1,[2]1,1,2)`

`a,b,[e,2]c` **~>** EAF`(b,c1,2,[2]1,[2].1,..2)` *:e*

`a,b.,[e,f]..c` *:d* **~>** EAF`(b,c1,2.,[2]1..,1..d)` *:f :e*

`a1,b2,[,,1]1 = a1,a1,[a,b]1` **≈>** EAF`(a,b1,[3]2)`

In our BTRIX matrix the next hyperspace entry is nested, while Bird's BEAF fills a plane of rows to its `,[3]` rim. Rules for EAF and BEAF were listed in last chapter, rules for BTRIX follow.

The hyper-row

BTRIX usurps a *hyper-row* – the structure called hyper-dimensional array by Chris Bird – to approach a multi-dimensional array structure as expressed in Bowers & Bird's BEAF.

The motor rule of Bowers (substituting the `b` predecessor expression for `b1`) dominates the results at this stage. But how do the other rules compare to Bird's hyperdimensional arrays?
Like Bowers, Bird employs a double system with arrays `a<X>b` in *Angle brackets* as intermediaries for the *Main level* reductions of BEAF. We have to sober up – there's a lot of redundancy in Bird's system. We brought his back to an EAF system, where only those methods effective in creating Bigger numbers are kept.
Apart from motor rule **3.** the BTRIX algorithm will resemble EAF. The theory is easy – in EAF and BTRIX we assume the upsize of subordinate structures to trickle down anyway. First by `a` (after the top structure is expanded by `b`), but later these lower structures will be upsized from significantly larger baskets `b` than before, so there is no loss.

List the rules of the BTRIX system up to linear hyperdimensions. Here an array wildcard `S` (and likewise `T` partial arrays) stands for a nested s_j`,..` *:m* row of entries.
The removal of trailing separators by rule **8.** is optional (leaving them is not significant, because any next size expansions by `b'` are accumulatively Bigger). In case rule **6.** doesn't find the dimension counter up-front (because there's a comma instead), the new rule **5.4** uploads the value from basket `b` to the right free nested entry. Later it will appear that this is just the nested version of ground level rule **5.3** and that both blend together in rule **5.6** for nested separators to arbitrary depth.

1.4 `a,b.,[`S_i`]..` *:k* `= a,b = b` (choice)

2.3 `a,Y,[s]{k1} = a,Y]{k}` (outer drop)

3.2 `a,b,1Z = a,ab,Z` (motor)

4.3 `a,.,[`S_i`]..1Z` *:k≥1* `= a,a.,[`S_i`]..Z` *:k* (fill)

5.3 `a,b1.,[`S_i`]..,1Z` *:k≥1* `= a,.,[`S_i`]..1b,Z` *:k*
`= a,a.,[`S_i`]..b,Z` *:k* (main upload)

5.4 `a,b1`$\underline{\$}$`,[,{m1}1T]1Z where` $\underline{\$}$ `≡ ,[`S_i`]..` *:k≥0*
`= a,a`$\underline{\$}$`,[,{m}b,T]1Z` (hyper-row upload)

6.3 `a,b`$\underline{\$}$`,[1T]1Z where` $\underline{\$}$ `≡ ,[`S_i`]..` *:k≥0*
`= a,a`$\underline{\$}$`.,[T]..1,[1T]Z` *:b* (main upsize)

7.1 `,[] ≡ ,`

8.1 `,[S],[T] {S<T} ≡ ,[T]` (weighted drop)

In rules like **5.4** (upload into nests) it is convenient to refill the empty basket `b` at once.
The stricter fill by rule **4.** results in larger numbers (decrementing `1Z` at ground level instead of uploading `b1` to `b` in the nest). But we'd run into notational problems for expressions `a,a,[,{m}b,T]z` when `m>z` and the nested array is uploaded until it has to drop off while `z` is counted down. Even though the usual reduction train `z` is last uploaded by rule **5.3** and very much Bigger than `m` when hyper-upload rule **5.4** kicks in, a lot of nested expressions would go unaccounted for – all in vain!
We want to make sure all sound-looking expressions are valid. We will forget rule **4.** in future definitions of BTRIX, and let rules **5.** handle the refill (of apple `a` in basket `b`) directly.

Because she is such a slow starter, hyperdimensional arrays in BTRIX have structurally smaller results than the same arrays in EAF. Fathom the effect of expanding arrays in both systems.

`a,a,[,,1]` Row `≡ a,b,[,,1],[1]1` **≈>** `EAF(a,a,b,[3]2)`

`a,a,[,,1]` Dim `≡ a,b,[,,1],[,1]1` **≈>**

`EAF(` Row `,[3]2) ≡ EAF(a,b,[2]2,[3]2)`

`a,b,[,,1],[1,1]1` **≈>** `EAF(a,b,[2]1,2,[3]2)`

`a,b,[,,1],[2,1]1` **≈>** `EAF(a,b,[2]1,1,2,[3]2)`

`a,b,[,,1],[,2]1` **≈>**

`EAF(a,b,[2]` Row `,[3]2) ≡ EAF(a,b,[2]1,[2]2,[3]2)`

`a,b,[,,1],[,,1]1` **≈>** `BEAF(a,b,[3]3)` **≈>** `EAF(a,b,[3]3)`

The values of ground level vars `a` and `b` don't matter anymore. The comparison is primarily about the structure and values of the separator arrays here. Notice that repetition of the highest separator in BTRIX compares to incrementation of the rightmost entry in EAF.

`a,b,[1,,1]1 = a,a.,[,,1]..1 :b` **≈>** `EAF(a,b,[3]1,2)`

`a,b,[1,,1],[1,,1]1` **≈** `EAF(a,b,[3]1,3)`

`a,b,[2,,1]1` **≈** `EAF(a,b,[3]1,1,2)`

`a,b,[,1,1]1 = a,a,[b-,,1]1` **≈** `EAF(a,b,[3]1,[2]2)`

`a,b,[1,1,1]1` **≈** `EAF(a,b,[3]1,[2]1,2)`

`a,b,[,2,1]1` **≈** `EAF(a,b,[3]1,[2]1,[2]2)`

`a,b,[,,2]1` **≈** `EAF(a,b,[3]1,[3]2)`

`a,b,[,,,1]1` **≈** `EAF(a,b,[4]2)`

`a,a,[` Row `]1 ≡ a,a,[,{b}1]1 = a,b,[,[1]1]1` **≈>**

`EAF(` Dim `) ≡ EAF(a,a,[b]2)` **≈** `BEAF(a,b,[1,2]2)`

Slow working BTRIX requires a full Row row of separator index entries (nested at level `1`), while EAF fills up Dim multiple dimensions (still at level `0`).

In the next section the structures Row and Dim of both nested systems appear to alternate.

Calculate norms in NIX for the separators of the first hyper-entry. Continue from the norm `n^^2` for multiple dimensions. Note that star `*` operators resolve by minority precedence: smaller first.

```
,[1,1] ≅ n^n*n = n**n1
,[,2] ≅ n^n*n^n = n**n*2
,[,3] ≅ ,[,2]*n^n ≅ n**n*3
,[,,1] ≅ n**n**2
,[,1,1] ≅ n**n*n1
```

Follow the norms `Ñ` for separators of the BTRIX hyper-row from this section.
When convenient mix stars with up-arrows `^` which have higher & majority precedence.

```
,[,,2] ≅ ,[,,1]*n^n^2 ≅ n**n*n*2
,[,,,1] ≅ n**n**3
,[,,,2] ≅ ,[,,,1]*n^n^3 ≅ n**n*n*n*2
,[,,,,1] ≅ n**n**4
,[,[1]1] ≅ n***3
```

Our NIX collapse of the hyper-row puts a power `n***2+**n` on top of the norm for multiple dimensions. Similarly, the hyperdimensional row in Bird's classification of separator spaces has level `ω^ω` which is a power `ω` above his class `ω` for multidimensional BEAF arrays.

Dimensional separators

Define special rules for the level of nested `,[,[n]]` multiple dimensions in BTRIX. There is no other place to go – we have to find room inside her separator subarrays.

let $\$$ ≡ ,[s_i].. *:k* $\$'$ ≡ ,[s_i].. *:m*

5.5 a,b1 $\$$,[$\$'$,1T]1Z (hyperdim upload)

= a,a $\$$,[$\$'$ b,T]1Z

6.4 a,b $\$$,[$\$'$,[1t]1T]1Z (hyperdim upsize)

= a,a $\$$,[$\$'$.,[t]..1,[1t]T]1Z *:b>0*

These rules will be wrapped up with earlier versions, in the section on general nested separators. Continue our comparisons from the last equation, where the *hyper-row size* of BTRIX rivalled the *number of dimensions* in EAF.

a,a,[,[1]1] Row ≈ EAF(a,a, Var,[1,2]2)

a,a,[,[1]1] Dim ≡ a,b,[,[1]1],[,1]1 ≈

EAF(Row,[1,2]2) ≡ EAF(a,b,[2]2,[1,2]2)

a,b,[,[1]1],[,,1]1 ≈ EAF(a,b,[3]2,[1,2]2)

```
a,a,[,[1]1],[Row]1 ≡ a,b,[,[1]1],[,[1]1]1 ≈
    EAF(Dim,[1,2]2) ≡ EAF(a,b,[1,2]3)
a,b,[1,[1]1]1 ≈ EAF(a,b,[1,2]1,2)
a,b,[,1,[1]1]1 ≈ EAF(a,b,[1,2]1,[2]2)
a,b,[,,1,[1]1]1 ≈ EAF(a,b,[1,2]1,[3]2)
a,a,[Row,[1]1]1 ≡ a,b,[,[1]2]1 ≈
    EAF(a,a,[1,2]Dim) ≡ EAF(a,b,[1,2]1,[1,2]2)
```

Again BTRIX hyper-row and EAF dimensions are comparable. Disparate structures that live on different levels? Or can we already see the two systems coming together?

```
a,b,[,[1]2],[,[1]2]1 ≈ EAF(a,b,[1,2]1,[1,2]3)
a,b,[1,[1]2]1 ≈ EAF(a,b,[1,2]1,[1,2]1,2)
a,b,[,1,[1]2]1 ≈ EAF(a,b,[1,2]1,[1,2]1,[2]2)
a,b,[,[1]3]1 ≈ EAF(a,b,[1,2]1,[1,2]1,[1,2]2)
a,b,[,[1],1]1 ≈ EAF(a,b,[2,2]2)
a,b,[,[1],2]1 ≈ EAF(a,b,[2,2]1,[2,2]2)
a,b,[,[1],,1]1 ≈ EAF(a,b,[3,2]2)
a,b,[,[1],[1]1]1 ≈ EAF(a,b,[1,3]2)
```

Notice the symmetry – fed back from the end of the hyper-row – repeating the dominant nested separator in BTRIX compares to incrementing the right nested array entry in EAF.

```
a,b,[,[2]1]1 ≈ EAF(a,b,[1,1,2]2)
a,b,[,[1],[2]1]1 ≈ EAF(a,b,[1,1,2]1,[1,2]2)
a,b,[,[2],[1]1]1 ≈ EAF(a,b,[1,2,2]2)
a,b,[,[2],[2]1]1 ≈ EAF(a,b,[1,1,3]2)
a,b,[,[3]1]1 ≈ EAF(a,b,[1,1,1,2]2)
a,a,[Dim]1 ≡ a,b,[,[,1]1]1 ≈
   EAF(a,a,[Row]2) ≡ EAF(a,b,[1,[2]2]2)
```

A multidimensional matrix Dim nested (end level 1) in BTRIX equals a subarray Row nested (middle level 1) in EAF. This reminds us how dimensional BTRIX matched the EAF linear array (both at ground level 0). Is our matrix system catching up?

Continue our NIX classification of structures, from norm n^^3 of the hyper-row. We found that moving a dominant first level nested entry 1 to a value of 2 is squaring the norm.

```
,[,[1]2] ≅ (n^^3)^2 = n^(n^n*2)
,[,[1],1] ≅ n^n^n1
,[,[1],2] ≅ n^(n^n1*2)
,[,[1],,1] ≅ n^n^n2
,[,[1],[1]1] ≅ n^n^(n*2)
```

We represent expressions of BTRIX by their dominant separator on the left. For suppose `n` is a Big number that upsizes the highest structure, its substructures will later be upsized `ñ` Big. In this BTRIX compares to BEAF array structures, exploded to a fixed size `n` on all levels at once. And then, if we let their separators drop by NIX, the norms `Ñ` on the right count up all entries `ñ` left.

```
,[,[2]1] ≅ n**n**n**2
,[,[2],[1]1] ≅ n**n**n*n1
,[,[2],[2]1] ≅ n**n**n*n*2
,[,[3]1] ≅ n**n**n**3
,[,[,1]1] ≅ n***4
```

Our norm for separator dimensions appends a power `,[,[1]1]+**n` on top of the hyper-row norm. Similarly, in Bird's classification he would now reach level `ω^ω^ω` above his `ω^ω` hyper.

Twice nested row

Take our comparison down to a deeper nested (level `2`) separator array. Refer to the general definition of BTRIX and EAF (for Bird's BEAF) for nesting depth in the next section.

Effectively we come to nest a *preceding expression* in BTRIX in the same basket `b` as in EAF, but the countdown occurs on the right in BTRIX, not on the left as in EAF.
Therefore appending the next structure (eventually a new Dim on the right) to a BTRIX expression, is comparable to building from the start (a new Row on the left) inside EAF.

```
a,a,[,[,1]1]Dim  ~  EAF(Row,[1,[2]2]2)

a,a,[Row,[,1]1]1  ≈  EAF(a,a,[1,[2]2]Dim)

a,a,[Dim,[,1]1]1 ≡ a,b,[,[,1]2]1  ≈

   EAF(a,a,[1,[2]2]1,[Row]2) ≡ EAF(a,b,[1,[2]2]1,[1,[2]2]2)

a,a,[,[,1],1]1  ≈  EAF(a,b,[2,[2]2]2)

a,b,[,[,1],,1]1  ≈  EAF(a,b,[3,[2]2]2)

a,b,[,[,1],[1]1]1  ≈  EAF(a,b,[1,2,[2]2]2)

a,b,[,[,1],[2]1]1  ≈  EAF(a,b,[1,1,2,[2]2]2)

a,a,[,[,1]Dim]1 ≡ a,b,[,[,1],[,1]1]1  ≈

   EAF(a,a,[Row,[2]2]2) ≡ EAF(a,b,[1,[2]3]2)
```

Again BTRIX hyper-dimensions and an EAF hyper-row are similar in a similar position.
Again the symmetry, that repeating the dominant separator in BTRIX (at a certain level of nesting), compares to rightmost incrementation (at the same level) in EAF. Vise versa, when the dominant

separator is repeated in EAF, the (dominant) rightmost entry nested at the next (deeper) level in BTRIX is incremented, as appears below.
The two systems show enough overlap to suggest they will soon merge together.

```
a,a,[,[1,1]Dim]1  ≈  EAF(a,a,[Row,[2]1,2]2)
a,b,[,[2,1],[,1]1]1  ≈  EAF(a,b,[1,[2]2,1,2]2)
a,b,[,[3,1]1]1  ≈  EAF(a,b,[1,[2]1,1,1,2]2)
a,b,[,[,2]1]1  ≈  EAF(a,b,[1,[2]1,[2]2]2)
a,b,[,[,,1]1]1  ≈  EAF(a,b,[1,[3]2]2)
a,a,[,[Row]1]1 ≡ a,b,[,[,[1]1]1]1  ≈
   EAF(a,a,[Dim]2) ≡ EAF(a,b,[1,[1,2]2]2)
```

A linear sub-subarray Row (middle level 2) in BTRIX equals a sub-dimensional array Dim (ends level 1) in EAF. The stand-off was similar at the end of the hyper-row section (one level ago). Because the Dim-Row alternation is as fast (over the first two levels) as the Row-Dim alternation, further nesting won't change their structural distance. This proves that BTRIX and BEAF run on a par over depth levels of nested separator arrays.

Continue our classification of structures from the norm n^^4 of first nested dimensional arrays.

```
,[,[,1]2] ≅ ,[,[,1]1]^2 ≅ n^(n^^3*2)
,[,[,1],1] ≅ ,[,[,1]1]^n ≅ n^(n^^3*n)
,[,[,1],[1]1] ≅ ,[,[,1]1]^n^n ≅ n^n^(n^n+n)
,[,[,1],[2]1] ≅ ,[,[,1]1]^n^n^2 ≅ n^n^(n^n+n^2)
,[,[,1],[,1]1] ≅ ,[,[,1]1]^n^^3 ≅ n^n^(n^n*2)
```

The added dimensional array ends the first batch. Now the second batch of this section. We use postponed operators again, with ++ to add to the top of operations past multiplication.

```
,[,[1,1]1] ≅ n**n**n**n1 = n***4++1
,[,[,2]1] ≅ n***4+*2
,[,[,,1]1] ≅ n***4+**2
,[,[,,,1]1] ≅ n***4+**3
,[,[,[1]1]1] ≅ n***5
```

We've extrapolated the last normations for the *twice nested row* from the previous level of the single nested *hyper-row* we've calculated at ~2 before. The newly nested row appends a power ,[,[1]1]+**n to our norm tower, while Bird reaches his ω^^4 class.

Further extrapolation to any *nesting depth* is easy enough. We quickly look up norms Ñ at comparable preceding separators. Cross level patterns emerge…

```
,[,[,[1]1]2] ≅ (n^n^n^n^n)^2
,[,[,[1]2]1] ≅ n^n^(n^n^n)^2
,[,[,[2]1]1] ≅ n^n^n^n^n^2
```

,[,[,[,1]1]1] ≅ n^^6

,[][1] ≡ ,.[,..1] *:n:* ≅ n^^(n*2) ≈ ñ****2

That last line gives an approximate norm ñ^^ñ for nesting depth. Bird did put his general nested arrays precisely at level ε_1 = ω^^(ω*2) in BIRD I (March 2012, page 4), but later classifies them at the same level ε_0 = ω^^ω as we do here (see any BIRD II, p1).
At this point in our algorithms, the difference between row and dimensional arrays blurs.

Nesting depth

Separators can be nested to arbitrary depth in BTRIX, with the general rules **5.** (value upload) and **6.** (structural upsize), to incorporate any previously defined nest level (main and nested).
This whole structure of nested array levels defines the first nesting or *single deep* .

let S_j ≡ ,[$S_{i,j}$].. *:r_j≥0* {0≤j<n}
(S depth index j init 0 to n− for *:n>0*)

5.6 a,b1.S_j,[..S_n,1T_n..]1T_j *:n:* (nestable upload b)
= a,a.S_j,[..S_nb,T_n..]1T_j *:n:*

6.5 a,b.S_j,[..1..T_j]1.Z *:n1:* (nestable upsize ,[T_n] by b)
= a,a.S_j,[..S_n.,[T_n]..1,[1T_n]..T_j]1.Z *:n :b>0 n:*

We operate at deeper levels by diving down the cascade of arrays of leftmost active separators. Separator arrays can be declared *active* in a rule, when directly followed ,[X]1t by a number entry on the right. At each nesting level along the cascade active separators are required, but only at the deepest level we count off the 1t (this is across levels in BTRIX always the leftmost).
Aside from the deepest active comma, deeper nested arrays may hide within inactive separators on the left (until reactivated), and in separators reached later on the right.

a,a,[,[Dim]1]1 ≡ a,b,[,[,[,1]1]1]1 ≈
EAF(a,a,[1,[Row]2]2) ≡ EAF(a,b,[1,[1,[2]2]2]2)

a,a,[,[,[Row]1]1]1 ≈ EAF(a,a,[1,[Dim]2]2)

The general comparison formula for deep nested BTRIX and EAF.
Apply evaluations ,≡,[] and ,≡,[1] to nest a level deeper (to *:n2:* after *Dim* cases).

a,a.,[..Dim..]1 *:n:* ≡ a,b.,[..,1..]1 *:n1:* ≈
EAF(a,a.1,[..Row..]2.) *:n:* ≡ EAF(a,b.,[1..1..]2.) *:n1:*

a,a.,[..Row..]1 *:n1:* ≡ a,b.,[..1..]1 *:n2:* ≈
EAF(a,a.1,[..Dim..]2.) *:n:* ≡ EAF(a,b.,[1..,2..]2.) *:n1:*

We have reached the point where separator nesting depth n can be expressed directly with a new depth index. Here BTRIX and BEAF fall together – for the same Big nesting depth in both systems the

resulting Big numbers can be considered approximately equal.

Where to attach any next deeps? After the subarray's opening bracket, and/or after or before its closing bracket? Or perhaps assign all to the separator comma in name?
Repetition of brackets may lead to a similar situation we had before, when we could have created a dimensional array with multiple commas `,..` *:m* but instead expressed it with a single entry subarray `,m]` or `,[m]`. A plane has room for `n` series of multiple commas, instead we wrote `,[m,n]` on the line. Soon to find out this notation was more economical.

Although Jonathan Bowers invented the nested array structures (see his legions arrays) as part of his BEAF, they were first defined properly in Chris Bird's nested array article.
Multidimensional arrays live at ground level, Bird's hyperdimensional arrays open up the `1`st level, and general nested arrays exist at the `j`th level of nesting.
We owe you the rules for nested arrays in EAF, whose growth rate is everywhere comparable to Bird's BEAF nested arrays (as argued in _2 before).

use $\$_j$ ≡ ,[1$S_{i,j}$]1.. *:$r_j \geq 0$*

iv. EAF(a,b,1,2Z) = EAF(a,a,b,1Z) **&'**

EAF(a,b.$\$_j$,[1..]2$Z_j$.) *:m1:* =

EAF(a,a.$\$_j$,[1..$\$_m$b,1$Z_m$..]2$Z_j$.) *:m:*

vi. EAF(a,b1.$\$_j$,[1..1X..]2$Z_j$.) *:m1:* =

EAF(a,a.$\$_j$,[1..$\$_m$.,[1X]1..1,[2X]1$Z_m$..]2$Z_j$.) *:m :b m:*

The comma `,[1]≡,` matches in the expression of rule `4'` as its deepest separator.
Rule `4'` can be seen as an initial case of rule `6` where `X≡-` collapses `,[]` to voids.

The fact that BEAF nests a complete predecessor expression in a common evaluation step, whereas BTRIX motor rule just applies addition, has finally lost its importance.
Can you guess why a rule for unlimited nesting of arrays (within an entry) and a system to nest an array within a separator [to a deeper level] have the same algorithmic strength? Take into account that an array can contain an unlimited series of separators!

While Bird soon represents his `[..] :k:` superseparator brackets by a new `\k` sign, we will increase the expressiveness of our brackets, to try to make the most of what we have.

BTRIX deep attachments

People take heed! How Beatrix advances in battle and beats legions beyond the ocean floor.

Dual deep

So far BTRIX only attached her nestable arrays `A,[X]Z` to the separator comma. Now she will express nesting depth, in a bold move to go beyond, by attaching brackets to brackets.
Jonathan Bowers already reached this algorithmic stage, with the Big Hoss for sure. Mathematical

historians may debate over Bowers' poorly defined system, how it counts separator nesting depth and where it fits in with Bird's and ours.
It won't be necessary to compare expressions of BTRIX with the Extended Array Functions of EAF and BEAF like we did before. If the rules across systems have comparable strength, and structures have similar data capacity, then the numbers expressed are approximately as Big.

The system BEAF of Bowers & Bird separates numbers between bracketed arrays (it helps their zero `0` is a sign). Their comma is strictly just an abbreviation sign for the subarray `[1]` (and if `[0]` existed here, it would fall away to separate units `1..` in entries).
In BTRIX and EAF a comma precedes separator subarrays (so number entries can be counted down until void `,,` in between commas).

In BTRIX an opening bracket `[` always *attaches* an array structure `S` to a preceding sign on the left. The separator comma `,[` had its inside space expanded with an index `s]` and then to linear and multiple dimensions and further `S]` through hyperspace, to the nested arrays of last chapter. Arrays are delimited by a corresponding `]` closing bracket on the right.

Watch the usual systems dig their initial first deep to an arbitrary Big nesting depth – working examples of how these algorithms can be implemented. Bird's isn't clear about the exact form of his rules in all cases, his articles have more to do with classifying higher structures than with the reduction of expressions to Big number.
Below we suppose his BEAF reduces superseparator `[[1]]` to a nested array of depth `b`, but we let the following `[[2X]]` repeat preceding superseparators at ground level. Better and Bigger is to bring all superseparators down to depth `b` too, as shown in EAF and BTRIX.

```
BEAF(a,2[[1]]2) = BEAF(a,2[1,2]2) = BEAF(a,a[2]a,a)
 <≈ EAF(a,3,[1[1]]2) = EAF(a,a,[1,[1,2]2]2)
      ≈ BTRIX[a,3,[][1]1] = a,a,[,[,1]1]1
          = a,a,[,[a-]1]1 = a,a,[.,[a--]..a-]1 :a-
BEAF(a,b[[2X]]2Z) ≈ BEAF(a,b.[[1X]]2..[[2X]]1Z) :b:
<<≈ EAF(a,b,[1[1X]]2Z) = EAF(a,a.,[1..[X]]2.,[1[1X]]1Z) :b:
    <<≈ BTRIX[a,b1,[][1X]1Z] = a,a,.[,..1][X].1,[][1X]Z :b:
```

In a comparison between systems similar wildcards `[X]` for subarrays are to be considered effectively equal.
Follows the rules for dual deep expressions `a,b,[V][W]Z` of BTRIX.

We use the REGEXP quantifier `?` for an optional "false/true" word. Below when $[S_{r,j}]?$ is "true" (then quantifier $[S_r]\{1\}$ matches at case `j`) this opens up the next deep nested array.
Instead of `([W])?` we dare to write `[W]?` to quantify the preceding 2nd subarray. There's no need to backslash `\[` in this context, and each particular `?` has the same truth value in the equation (think of signs `?` as indexed $?_{i,j}$ by the free indexes of the word it quantifies).

let $\underline{\$_k}$ ≡ $,[S_{i,k}][T_{i,k}]?..\ :r_k$ $\{\text{init}_{i=0}\ r \geq 0\}$

$\underline{¥_j}$ ≡ $\underline{\$_j},[S_{r,j}]?$

5.7 $a,b1.\ \underline{¥_j}[..\ \underline{\$_n},1..Y_j]1.Z\ :n:$ (upload counter)

```
        = a,a.¥j|[..$n|b,..Yj]1.Z :n:

6.6  a,b.¥j|[..$n|,[1V][W]?1..Yj]1.Z :n:    (upsize dimensions)
        = a,a.¥j|[..$n|.,[V][W]?..1,[1V][W]?..Yj]1.Z :n: :b

9.0  a,b.¥j|[..$n|,[][1W]1..Yj]1.Z :n:    (uproot nests)
        = a,a.¥j|[..$n|.,[..][W]1.,[][1W]..Yj]1.Z :n: :b:
```

Miraculously rule **6.** handles all size increases here, also those in the second deep. Rule **9.** is new, it "uproots" nestings of `b` levels, subtracting `1` from the active iterator (Bowers' pilot) past the dual deep (subtracting from the second deep itself would be significantly slower in the long run).

With hindsight, a nestable system with mixed separators (to allow alternative `,[M],[N]` constructs) would prove to be superfluous (from here on), now that serial deeps also "mix arrays".
A related question is, if combining two or three attachment styles would increase the power of the algorithm significantly, compared to deploying a new `/` type of separator and/or `{}` brackets, given that each new point of attachment expands the information space similarly.

There are different positions and methods to attach a next batch of arrays.
In BIRD I his *superseparators* are attached `[..X..] :k:` to *both sides* at once. That's before Bird contains them with a more advanced separator `[X\k]` sign (his backslash).
It's more profitable for an array system to attach deep arrays *left inside* `[..Xi] :n:` to the opening tag, or *right inside* `[Xi..] :n:` to the closing tag as in EAF, because we get an array space for free. In this extra space a deep entry over the superarray will have the same strength as the number of brackets on *both sides* in Bird's early system.
Note that in system EAF a *right inside* attached array contains the next deep. But, if we let our rules match strictly from left to right, a *left inside* attached array would be the superarray iterated over by the right serial deep (clear of, but denominated by the number of closing brackets on its left), as if dwelling on top of Bird's *both sides* subsystem.

A first choice in BTRIX is to have deep arrays in series attached *left outside* `,.[Xi].. :n` to the closing tag. Then we can hold the attachment *left inside* (of serial arrays) in reserve for the next batch, instead of introducing a new separator character (a future `/` big comma).
Again `,[V][k] ≡ [..V..] :k:` and dual deeps in BTRIX `,[V][W] ≡ [V\W]` versus Bird are comparable in capacity. Further serial deeps in BTRIX `,.[Vi].. ≡ [.Vi\..1] ≡ [1\\2]` have the same data capacity as a row of backslashed BEAF.
We won't follow in Bird's footsteps, since we are keen to know exactly where the already employed characters will be exhausted completely. Going to squeeze the maximum Big out of our current `[]` bracket pair.

Serial deeps

Writing rules as equations of general expressions has become rather cumbersome. The part that changes can be hard to find, but is all that matters in a rule.
We'll introduce a REPEXP "rewrite" style algorithm for *serial deeps* that is hopefully more legible. Our prescriptions should solely match the changing subsequence – with REPEXP right on target – like Bowers' airplane analogy for array structures made rigorous.

Variables `n≥0` are greedy for the unit signs `1..` they consist of. Here we use the unit `1` but it can

stand for any number sign (like 2,ω, etc), but not for zero 0 which is empty (counted down fully), and generally not for any number n<1 which cannot reach a limit when counted down.
In our RepExp the sign ! negates the following characters (up to the space), so !1 means not to match one. This exclamation ! sign can also function as a positioning dot.

Dots are used . to mark positions, so that we can insert ↓ something there. To insert a minus sign after a number n1 ↓ - means to count down to number n (decrement 1- ≡ 0 to empty).
We can move up ↑ a previous string or value b, so it is lifted from place and inserted at the dot(s where it would repeat *:b* a selected word some b times).
There are two search directions. We start moving (the cursor) to the right →Y until we find the first word Y, or by X→YXZ the first X which is contained in YXZ (other containments are covered by earlier listed rules).
We can also move back X←Y to get at word X in Y on the left. Now we have four *arrow moves*.

Series of separator arrays (nested to any depth) can be written with the signature ,[W] where start and end bracket correspond to the same *parent separator*. Those brackets need not be a pair, as would be the case with ,.[V_i]..1 serial deeps.
This way the rule for *serial deeps* in Btrix looks simpler in RepExp notation.

9.1 a,b .→,[V][1W]1↓- where parent nests
↓,[V..][W]1 *:↑b:* are all]1 active

A program that runs the algorithm with RepExp searches → the next counter (Bowers' *pilot*) directly right of a separator or its array, which can be declared *active* (ready to iterate over).
As before, in Btrix we have the provision that (for an array to be active) all the (serial) arrays it is embedded in must also be active (have a right iterator). Therefore (to put b to good use) in Btrix the whole parent context (an active array is nested in) should be checked (up to ground level) to affirm the form]1 at every (serial array) closing bracket (closer).
To do this the program moves right to find brackets] that are unaccounted for → passing serial sisters → counting (down and up) the depth of nieces → until it rises to the grand mothers.
If a (grand) parent level closer is met], followed by an inactive entry (counted down to void), we've established that the [W], parent (serial) array is *inactive*. Then Btrix moves on → from there, restarting the procedure (find a next candidate).
Only if all arrays in the pedigree have active]1 closers, any old rule of Btrix can be applied. For rule **9.** the character directly left of the opening [must (in Btrix!) be the] closing bracket of an empty array [] to insert a new nesting to depth b with copies of the preceding mother in.

Is it useless and inefficient to expand possible (other) occurences of active separators, when they are nested inside an inactive parent? Or is it actually a good strategy?
We can prove it doesn't do much harm, by resolving some extreme examples. Because expressions where this matters are extremely rare, the growth rate will not be significantly affected either way.

Let a<<b the basket be Big, for expressions of Btrix (by rule all grand parents must be active) versus system Betrix (only the direct parent must be active) to work out differently.
The x^b denotes an array whose structure is dominated by the value of the superscript, just like x^W is a variable dominated by the strength of the superscripted word.

a,b,[,[1V]1],1 case Btrix < Betrix
5. ≈ a,a,[,[1V]1]b, **6.** = a,a,[.,[V]..1,[1V]]b, *:a*

```
                = a,a,[X^a]b
    6·2.= a,a,[.,[V]1..],1 :b
          == a,a,[X^b]a  {X^a<<X^b}
                <≈ BTRIX[a,b,[,[1V]1]1,1]

a,b,[,1],[,1,W]1   case BETRIX < BTRIX
    =5.= a,a,[b],[a,,W]1
          = a,a,[b],[Y^a]1  {b<<x^W<<Y^a<<Y^b}
     5.≈ a,a,[,1],[b,,W]1
          == a,a,[x^W],[Y^b]1
                <≈ BETRIX[a,b,[,1]1,[,1,W]1]
```

These cases show ambiguous results (spread the risk). System BETRIX doesn't bother if her grandparents are active (as long as the direct separator or parent array is). She doesn't need to, because in a common reduction train such cases are insignificantly few.
Forms `],` would only occur after some dominantly Big `]b,` iteration was exhausted (1st example). And the upload over two forms `[,` is a rare event too (2nd example). This is confirmed by creating competing **<≈** successor expressions with no significant cost.

We do take care to remove trailing arrays from BETRIX though, after rule **6.** by rule **2.** which is defined later, as part of her core REPEXP regime. No defunct structure that gives no returns can be allowed to suck up the accumulated value of `b` (where initial value `a` comes in place). No way!

The REPEXP program to resolve (rewrite) expressions in BETRIX runs as follows:
Move right → until a number sign `1` is matched. If there's a comma `,` on its left, apply rule **3.** (to ↓ add `a`) or rule **5.** (to ↑ move `b` up). If it has an opening bracket `[` directly on its left, mark it, and proceed → to find the corresponding closing bracket `]` on its right. To identify bracket pairs a machine program has to keep score of the nesting level it is searching in.
If the current closer `]` is directly followed by the `1` of a counter, we are ready to act on this `[W]1` separator array. Else if the closing `]` is followed by an opening `[` find → the next corresponding `]` and loop → until the character right of this closer is either: a counter `1` (for the active `[W]1` serial array), or a separator character `,` (the counter is void).
For an *active* (serial) array, count down the entry `]1-` right after the closer. Go back ← to the marked opening `[` and match the character directly to its left, to determine which rule to apply.
If it's the very separator `,` rule **6.** applies (to copy the parent array `b` times in series). **UNDER CONSTRUCTION** Else in BETRIX (!) if an opening bracket `[` is matched rule **9.** applies (to nest copies of the preceding mother to depth `b`).
Else at an *inactive* (serial) array in BETRIX, (for her REPEXP rules to match) we set the cursor back ← to the opening mark and continue → to query its child arrays. Note that in BTRIX we searched right on after an inactive closer, and the resulting growth difference was provably insignificant, so that's an option too (though not strictly REPEXP protocol).

Past the single deep nested array, we see room to attach two complementary types of inside arrays and one extra type on the outside (as in BTRIX). But given the default matching direction of rules (left to right), it seems natural to first reduce superarrays attached *left inside*, then those *right inside* and only then (if other rules fail) those attached *right outside*.

Rules in BETRIX are written in REPEXP notation, and listed in order of execution.

Entries `a>0` and `b>0` form her double base.
The machine searches right `→1` for a number and establishes its position – to be either some `,1` counter in between, or at the start `[1` of an active separator array (with the counter right after).

0.2 BETRIX`[a, ≡ a,`
(show: without left function signature)

3.3 `a,b.,1↓- ↓a`
(motor: with counter `c` over `b`, append `a` to `b` and count down `c`)

5.8 `a,b, .→,1↓- ↑b`
(upload: counter down over an empty entry, and move `b` up here)

4.4 `a,., ↓a`
(free fill: if entry `b` is void put `a` in place without countdown)

`2.4` `,] ≡ ]` **2.5** `[W]] ≡ ]` **so** `,[W]] ≡ ]`
(separator drop: trailing entry @ **5.** and subarray @ **6.**)

UNDER CONSTRUCTION

6.7 `a,b .→,[1W]1↓-`
`↓,[W]1.. :↑b`
(upsize: counter down past separator, repeat pre-separator and `1`)

UNDER CONSTRUCTION

9.2 `a,b .→,[[1V]W]1↓-` `9.2` `{1∉V}`
`↓,[V..][W]1 :↑b:`
(uproot: counter down over serial deeps, dig nestings)

`7.2` `[]1 ≡ 1` `7.3` `,[-]1 ≡ 1`
(nest drop: trailing right array @ **6·9.** & fit to drop `,[-]` @ **5·6.**)

`1.5` `a,b →!1 = b` **1.5** **else** `a,b,X = b`
(choice: if there's no counter found, output `b`)

`4.5` `a,!1, ↓,`
(else fill: if `a` is a single base then insert extra entry)

8.1 `,[S],[T] {S<T} ≡ ,[T]` **OBSOLETE**
(weighted drop: Bird's option to remove “semi-trailing” separators)

The BETRIX uproot rule **9.** is pretty maximal, but different than in BTRIX.
Rules of allowance are not necessary for the complete reduction of an expression, and we mark this route in a less pronounced manner.
We won't use rule **8.** anymore. It has an insignificant effect on the resulting number, which is too Big already (to drop smaller left separators makes the result slightly smaller). Cleaning up useless separators may be a natural reflex, but we find the comparison rules for array sizes too difficult to implement properly for all types of arrays (see how Bird struggles).
So in our system we reuse (which merely “adds” to sizes) these semi-trailing separators.

Example expressions in BETRIX. The start is a little awkward but the superpower class of each linear entry is the same as in BTRIX.

```
3,,1  4.= 3,3,1  3.= 3,33,  1:= 6
3,,,1  4.= 3,3,,1  5.= 3,,3,  4.= 3,3,3,
       3.= 3,33,2,  =3.= 3,3333,,  1:= 12
3,[1]1 =4.= 3,3,[1]1  6.= 3,,[]1,[]1,[]1,[1]
       =7·2.= 3,,1,1,1  4·3:= 3,6,,1,1
       =5·4·3:= 3,21,,,1 =5·2·4·3:= 3,12,,20
       =5·4·3·1:= 47071589412 ≈ 5E10 ≈ 3^22.4
```

At the moment our BETRIX algorithm is like a Dutch cheese with series of deep nested holes.

Opening attachment

To separate serial deeps `[1]..` we'd like to sort them in *holes*, just like numbers `1..` were arranged in entries, rows, dimensions, nests and deeps. Now to line up a row of "hole entries" we need a new construct to separate our arrays with, but which?

To have a bigger comma `,` you might extend it to a slash `/` forward comparable to Bird's slash `\` backward (which takes off a hole-dimension sooner, as Bird reduces deeps to slash).
After the first hole-row $[X_{i,j}]../..$ you can have dimensions, nests and deeps of holes, by using a next type of bracket characters `/{}` filled with number entries `m`, comma nests `,[W]` and serial deeps (using ground separator chars). You'd hole-nest them `/{}` to arbitrary depth, and further iterate over serial hole-deeps `/.{1}..` to exhaust a similar batch of structures.
However, this will cost you a lot of characters before long!

We will use Bird's original construction "beyond nested arrays" to fill the 2nd batch of structures.

To have a bigger comma `,` we extend it to `,[[]1] = ,[[b]] = ,.[1].. :b`
After the first hole-row $[X_{i,j}]../..$ we have dimensions, nests and deeps of holes, by using a next type of bracket characters `/{}` filled with number entries `m`, comma nests `,[W]` and serial deeps (using ground separator chars). We hole-nest them `/{}` to arbitrary depth, and further iterate over serial hole-deeps `/.{1}..` to exhaust the 1st batch of structures.

Big comma

Betrix numerous characters

Instead of defining each batch of structures anew, with perhaps a dedicated comma sign and two new brackets (the theoretical minimum would be one new bracket sign), we like to address each batch `¢(x)` with an index. So we can have a new `,(2) ≡ \` separator sign, and `](1) ≡ }` as above, and we started `[() ≡ [` with a 0th batch of characters.

To differentiate between char-index brackets and common round brackets (not used in resolving expressions in BTRIX), we may prefix an empty sign `0(X)` to the latter (or just rely on context).

The algorithms that breathe life into this world of indexed characters should be so similar for each, that a general definition for arbitrary batches of structures is not too far fetched.
Just imagine what would happen if we *upgrade* a character `¢(,1Y)` with value `b` from the basket. For example:

Beatrix record maxima

Bigger

In theory we can measure the "whole set" of counting numbers (whatever that may be) with the countable infinity `ω`. This omega could very well mark the end of our efforts to define Big numbers by recursion. Recursion is the repetition of a rule or function, here we've applied the primitive rule that adds `1` to create the successor of every number.
However, the first infinity `ω` is the root number of the flourishing branch of set theory. It is only the beginning of some infinitely Bigger numbers.

Bigger jumps past infinity

Allow the unit `1` to be subscripted `1[1] = ω` and start a new nested function (as above).

Plan β is to reach for an expert

Biggest

We don't think any mathematical definition of infinity could be the final word that the finest chicken brains of the world (versus the geniuses under the intergalactic hood) can come up with. In our experience future mathematicians, dependent on the available resources, will always invent new concepts and constructions to replace the old.

Biggest supralogical fantapsy

Suppose we formalize our mind egg in a self-reflective function, which boldly generates new generations of generalized mathematicians. Let the Biggest number the initial generation comes up with quantify the mathematical resources (for example financially ;-) the next virtual generation of mathematicians can work with to define their Biggest number, and so on.

We assume any Bigger than infinite amount of resources is available for God's project (were Sarasvatî to rule over science, her devotees must grant us those). Likewise the number of maths generations may be determined by a previously Biggest variable fed back to its hyper-recursive system. Mind you, this very meta-system structure is adapted with every new generation (whatever that entails, a logician like Smullyan will come up with something).
Is the Biggest number forced to grow / adapt? Or is there an axiomatic paradox, such as `0=1` suggested by Kanamori in *The Higher Infinite*, so that the whole process must stop at some odd point of no return? Eh, eh, even when counting in an infinite host of future Columbuses!?
And what if we postulate a number so high that it cannot be reached by this fantastic generator function of mathematicians xor philosophers? A psychic paradox `Ψ` that lies wholy beyond?!

Tablature

Postponed operators

Hands on! We evaluate `a*{c}b` superpower stars by putting postponed operations `+*{s}t` right on top. The prefixed plus sign postpones the right superstar operation until the same number of stars occur on the left. This enhanced system has a better resolution, useful for comparisons.

Define how *postponed superstars* work. Use `{k}` RepExp to repeat signs, so that for example `2*{4}3 = 2****3` whose reduction train is shown in the left column of the table below.
The underscore _ signifies an arbitrary part, not changed by the operation.

```
a^··b ^:c = a*··b *:c1 = a*{c1}b = a,{c2}b

_a*{c1}b1 = _a*{c1}b+*{c}a

_a*{c1}b1+*{s}t = _a*{c1}b+*{c}a+*{s}t

_+*{c}a+*{s}t = _+*{c}a*{s}t

_a*{c}1+_ = _a+_
```

Postponed addition in `a*{c>1}b++t` benefits from an extra plus sign, to distinguish it from `a*b+t` which works like ordinary addition (in the table below).
Expressions of the first row of Btrix (abbreviated to *Bx* in the table head) is translated to postponed superstars format in the column left.
An alternative system *By* is presented in the right columns, its iterator entries are counted down to `1` (instead of `0`) and *By* can be read as a series of up-arrow operations.

Calculation of `11,,,,,111 = 1.. :65536`				
	**	Bx to ,0	^	By to ,1
1	2****3	2,,,,,3	2^^^3	2,3,1,1,1,1,2
2	2****2+***2	2,2,,,,2	2^^2^^2	2,1,1,1,1,4,1
3	2****1+***2***2 2+***2***2	2,,,,2,1		2,2,1,1,1,3
4	2+***2***1+**2 2+***2+**2	2,2,,,1,1	2^^(2^2)	2,2,1,1,2,2
5	2+***2**2	2,,,2,,1		
6	2+***2**1+*2 2+***2+*2	2,2,,1,,1	2^^(2*2)	2,2,1,2,1,2
7	2+***2*2	2,,2,,,1		
8	2+***2*1+2 2+***2+2	2,2,1,,,1	2^^(2+2)	2,2,2,1,1,2
9	2+***4	2,4,,,,1	2^^4	2,4,1,1,1,2
10	2***4	2,,,,4,		
11	2***3+**2	2,2,,,3,	2^2^2^2	2,1,1,1,5,1
12	2***2+**2**2	2,,,2,2,		2,2,1,1,4
13	2***2+**2**1+*2 2***2+**2+*2	2,2,,1,2,	2^2^(2*2)	2,2,1,2,3
14	2***2+**2*2	2,,2,,2,		
15	2***2+**2*1+2 2***2+**2+2	2,2,1,,2,	2^2^(2+2)	2,2,2,1,3
16	2***2+**4	2,4,,,2,	2^2^4	2,4,1,1,3
17	2***1+**2**4 2+**2**4	2,,,4,1,		
18	2+**2**3+*2	2,2,,3,1,	2^(2*2*2*2)	2,2,1,4,2
19	2+**2**2+*2*2	2,,2,2,1,		

20	2+**2**2+*2*1+2 2+**2**2+*2+2	2,2,1,2,1,	2^(2*2*(2+2))	2,2,2,3,2
21	2+**2**2+*4	2,4,,2,1,		
22	2+**2**1+*2*4 2+**2+*2*4	2,,4,1,1,	2^(2*2*4)	2,4,1,3,2
23	2+**2+*2*3+2	2,2,3,1,1,	2^(2*(2+2+2+2))	2,2,4,2,2
24	2+**2+*2*2+4	2,4,2,1,1,	2^(2*(2+2+4))	2,4,3,2,2
25	2+**2+*2*1+6 2+**2+*2+6	2,6,1,1,1,	2^(2*(2+6))	2,6,2,2,2
26	2+**2+*8	2,8,,1,1,	2^(2*8)	2,8,1,2,2
27	2+**2*8	2,,8,,1,		
28	2+**2*7+2	2,2,7,,1,	2^(.2+..2) *:7*	2,2,8,1,2
29	2+**2*6+4	2,4,6,,1,	2^(.2+..4) *:6*	2,4,7,1,2
30	2+**2*5+6	2,6,5,,1,	2^(.2+..6) *:5*	2,6,6,1,2
31	2+**2*4+8	2,8,4,,1,	2^(.2+..8) *:4*	2,8,5,1,2
32	2+**2*3+10	2,10,3,,1,	2^(2+2+2+10)	2,10,4,1,2
33	2+**2*2+12	2,12,2,,1,	2^(2+2+12)	2,12,3,1,2
34	2+**2*1+14 2+**2+14	2,14,1,,1,	2^(2+14)	2,14,2,1,2
35	2+**16	2,16,,,1,	2^16	2,16,1,1,2
36	2**16	2,,,16,,		
37	2**15+*2	2,2,,15,,	2*..2 *:15*	2,1,1,17,1
38	2**14+*2*2	2,,2,14,,	2*..2*2 *:14*	2,2,1,16
39	2**14+*2*1+2 2**14+*2+2	2,2,1,14,,	2*..(2+2) *:14*	2,2,2,15
40	2**14+*4	2,4,,14,,	2*..4 *:14*	2,4,1,15
41	2**13+*2*4	2,,4,13,,	2*..2*4 *:13*	
42	2**13+*2*3+2	2,2,3,13,,	2*..(2+2+2+2) *:13*	2,2,4,14
43	2**13+*2*2+4	2,4,2,13,,	2*..(2+2+4) *:13*	2,4,3,14
44	2**13+*2*1+6 2**13+*2+6	2,6,1,13,,	2*..(2+6) *:13*	2,6,2,14
45	2**13+*8	2,8,,13,,	2*..8 *:13*	2,8,1,14
46	2**12+*2*8	2,,8,12,,	2*..2*8 *:12*	
47	2**12+*2*7+2	2,2,7,12,,	*(2*){12}*((2+){7}2)	2,2,8,13
48	2**12+*2*6+4	2,4,6,12,,	*(2*){12}*((2+){6}4)	2,4,7,13
49	2**12+*2*5+6	2,6,5,12,,	*(2*){12}*((2+){5}6)	2,6,6,13
50	2**12+*2*4+8	2,8,4,12,,	*(2*){12}*((2+){4}8)	2,8,5,13
51	2**12+*2*3+10	2,10,3,12,,	*(2*){12}*(2+2+2+10)	2,10,4,13
52	2**12+*2*2+12	2,12,2,12,,	*(2*){12}*(2+2+12)	2,12,3,13
53	2**12+*2*1+14 2**12+*2+14	2,14,1,12,,	*(2*){12}*(2+14)	2,14,2,13
54	2**12+*16	2,16,,12,,	2*..16 *:12*	2,16,1,13
55	2**11+*2*16	2,,16,11,,	2*..2*16 *:11*	
56	2**11+*2*15+2	2,2,15,11,,	*(2*){11}*((2+){15}2)	2,2,16,12
57	2**11+*2*14+4	2,4,14,11,,	*(2*){11}*((2+){14}4)	2,4,15,12
58	2**11+*2*13+6	2,6,13,11,,	*(2*){11}*((2+){13}6)	2,6,14,12
∴				
68	2**11+*2*3+26	2,26,3,11,,	*(2*){11}*((2+){3}26)	2,26,4,12
69	2**11+*2*2+28	2,28,2,11,,	*(2*){11}*((2+){2}28)	2,28,3,12
70	2**11+*2*1+30 2**11+*2+30	2,30,1,11,,	*(2*){11}*((2+){1}30) *(2*){11}*(2+30)	2,30,2,12
71	2**11+*32	2,32,,11,,	2*..32 *:11*	2,32,1,12
72	2**10+*2*32	2,,32,10,,	2*..2*32 *:10*	
73	2**10+*2*31+2	2,2,31,10,,	*(2*){10}*((2+){31}2)	2,2,32,11
74	2**10+*2*30+4	2,4,30,10,,	*(2*){10}*((2+){30}4)	2,4,31,11
∴				
102	2**10+*2*2+60	2,60,2,10,,	*(2*){10}*((2+){2}60)	2,60,3,11
103	2**10+*2*1+62			

	2**10+*2+62	2,62,1,10,,	*(2*){10}*(2+62)	2,62,2,11
104	2**10+*64	2,64,,10,,	2*..64 *:10*	2,64,1,11
105	2**9+*2*64	2,,64,9,,	2*..2*64 *:9*	
106	2**9+*2*63+2	2,2,63,9,,	*(2*){9}*((2+){63}2)	2,2,64,10
∴				
168	2**9+*2+126	2,126,1,9,,	*(2*){9}*(2+126)	2,126,2,10
169	2**9+*128	2,128,,9,,	2*..128 *:9*	2,128,1,10
170	2**8+*2*128	2,,128,8,,		
∴				
298	2**8+*256	2,256,,8,,	2*..256 *:8*	2,256,1,9
∴				
555	2**7+*512	2,512,,7,,	2*..512 *:7*	2,512,1,8
∴				
1068	2**6+*1024	2,1024,,6,,	2*..1024 *:6*	2,1024,1,7
∴				
2093	2**5+*2048	2,2048,,5,,	2*..2048 *:5*	2,2048,1,6
∴				
4142	2**4+*4096	2,4096,,4,,	2*..4096 *:4*	2,4096,1,5
∴				
8239	2**3+*8192	2,8192,,3,,	2*..8192 *:3*	2,8192,1,4
∴				
16432	2**2+*16384	2,16384,,2,,	2*..16384 *:2*	2,16384,1,3
∴				
32817	2**1+*32768 2+*32768	2,32768,,1,,	2*..32768 *:1* 2*32768	2,32768,1,2
32818	2*32768	2,,32768,,,		
32819	2*32767+2	2,2,32767,,,	2+..2 *:32767*	2,0,32769,1 2,2,32768
32820	2*32766+4	2,4,32766,,,	2+..4 *:32766*	2,4,32767
∴				
65584	2*2+65532	2,65532,2,,,	2+..65532 *:2*	2,65532,3
65585	2*1+65534 2+65534	2,65534,1,,,	2+65534	2,65534,2
65586	65536	2,65536,,,,	65536	2,65536,1 2,65536
65587		65536		65536