FAQ : SSS Enumeration

Introduction

Enumerations are useful, both theoretically and practically. The existence of a set enumeration guarantees that the set is at most countably infinite. For example, an enumeration of the rationals proved that there are the same number of fractions as integers, while a proof that no enumeration of the reals exists showed that the real numbers are uncountable. More usefully, an enumeration assigns an index to every member of the set under consideration, giving a practical means to consider every case. This makes enumeration a powerful part of the methodology found in NKS (“New Kind of Science”).

Sequential substitution systems (SSS or “sessies”) are defined by sets of rules (here called "rulesets"), each consisting of a target string and a replacement string. Given some initial state (which may also be represented as a string) these rules are applied and the system evolves. But without a well-defined enumeration of strings and rulesets any treatment of sequential substitution systems will be haphazard and may miss important features. To develop an enumeration of all sessie rulesets, it was first necessary to enumerate all strings and then all lists of all strings. These enumerations can be used or modified for other applications based on rulesets and initial state strings (for example, non-sequential substitution systems, multiway systems, etc.).

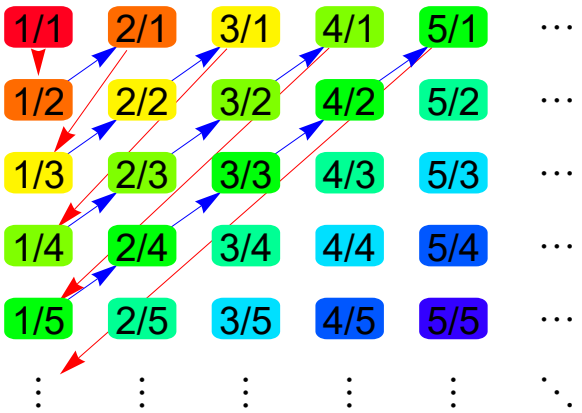
In developing an enumeration, previous successful enumerations can be studied to good advantage. More detail on these initial steps can be found here (from which some of the following summary is taken):

> http://www.mathematica-journal.com/2011/05/indexing-strings-and-rulesets/

Enumeration of all Rationals

Cantor’s Diagonalization

There are ways to create an ordered list of things that grow infinitely in two different “directions”. One is Georg Cantor's famous diagonal ordering of the rational numbers:



Both the numerator and the denominator of the fraction are taken from an infinite (but countable) set, and rather than trying to treat one infinity first, as in {1/1, 2/1, 3/1, … 1/2, 2/2, 3/2, …, 1/3, 2/3, 3/3, … }, this method allows growth in both directions to continue indefinitely, following a defined pattern while clearly including all possible combinations. One drawback of this method as applied to fractions is that equivalent fractions get counted multiple times. For example 1/1 = 2/2 = 3/3 = …, 1/2 = 2/4 = 3/6 = …, etc. Another (possible?) drawback is that for a given pair of integers {a,b}, a/b occurs in the list far earlier than b/a.

See also:
https://demonstrations.wolfram.com/EnumeratingTheRationalNumbers/
https://demonstrations.wolfram.com/APathThroughTheLatticePointsInAQuadrant/

The mathematics literature contains many examples of non-repetitive ways of ordering the rationals, but none of the non-repetitive sequences has the simple clarity of the diagonal arrangement. Is it so bad to have duplicates and then be forced to ignore or drop them later? This is an important question that will return in various situations. Although a little inelegant, the existence of duplicates hurts nothing essential, so we'll consider non-repetition a desirable but not necessary feature.

What is essential then?

1. The non-ambiguity of the list: it can be generated out to any desired number of elements, and the order can be unambiguously described. Here the fractions are listed in increasing order first by *sum* of numerator and denominator (as colored in the figure above), next by *numerator*.

2. The existence of a *successor* algorithm: from a given fraction n/d, can the next fraction in the list be found? Yes, if $d > 1$, the next fraction is $(n + 1)/(d - 1)$, else if $d = 1$, it is $1/(n + 1)$. This means that we do not need to generate the whole list at once, we can proceed one step at a time, perhaps testing or making some use of the fractions as they are generated. A small modification to the successor algorithm lets us easily bypass duplicates: if the successor found is not a fraction reduced to lowest terms, advance to *its* successor.

3. The existence of *rank* and *unrank* functions, to convert back and forth between the list and an ordered list of integers. (Given such functions the definition for *successor*[*element*] might be as

simple as *unrank*[*rank*[*element*]+1], if no direct method of advancing through the enumeration has been found.) For the diagonal ordering, we must determine which diagonal we want and then which element. The fraction n/d appears on the $(n + d - 1)^{th}$ and is element n on that diagonal. An easy way to do this is create a function to generate the n^{th} triangular number (the total number of entries in the previous diagonals). For an implementation of successor, rank and unrank for the Cantor enumeration of the rationals, see the paper referenced above.

Nonrepetitive Indexing of the Rationals

... based on Prime Factorization

As mentioned above, there actually are 1-1 and onto (bijective) mappings between the set of rationals (either all rationals or the positive rationals) and $\mathbb{Z}^+$, the set of positive integers. One elegant algorithm relies on the Fundamental Theorem of Arithmetic (also known as the Unique-Prime-Factorization Theorem): any integer greater than 1 can be written as a unique product of prime numbers (up to ordering of the factors). For example,

In[50]:=
```
showFactorization = Row[{#, " == ", Row[Superscript @@@ FactorInteger[#], "×"]}] & ;
showFactorization[3 083 080]
```

Out[51]=

$$3\,083\,080 \;==\; 2^3 \times 5^1 \times 7^2 \times 11^2 \times 13^1$$

If the prime numbers are listed in order, the sequence of exponents provides a unique way to characterize each positive integer. For the above example the sequence is {3, 1, 2, 2, 1, 0, 0, 0, …}. But the same can be said of all possible numerators and denominators of rational numbers, and furthermore, when a fraction is reduced to lowest terms no prime factor will appear in both the numerator and denominator, a fact which motivates the following algorithm in which odd exponents define factors of the numerator, even exponents the denominator:

In[52]:=
```
IntegerToRationalByFactorization[n_Integer] :=
 Times @@ (#1^If[EvenQ[#2], -#2/2, (#2+1)/2] & @@@ FactorInteger[n])
```

```
IntegerToRationalByFactorization[2^5 × 3^4 × 5^3 × 7^2 × 11^1]
```

$$\frac{2200}{63}$$

In[ ]:=
```
showFactorization[2200 / 63]
```

Out[ ]=

$$\frac{2200}{63} \;==\; 2^3 \times 3^{-2} \times 5^2 \times 7^{-1} \times 11^1$$

Note that even exponents in the factored integer are halved and then used as the exponents of the same prime factors in the denominator, odd exponents are halved, rounded up and similarly used to specify the numerator. Of course this procedure is not unique, the treatment of odd and even expo-

nents could just as well be reversed. A disadvantage of this method of ordering the rationals is the order itself: it preferentially treats integers (and in general, small denominator fractions) before others.

```
IntegerToRationalByFactorization /@ Range[50]
```

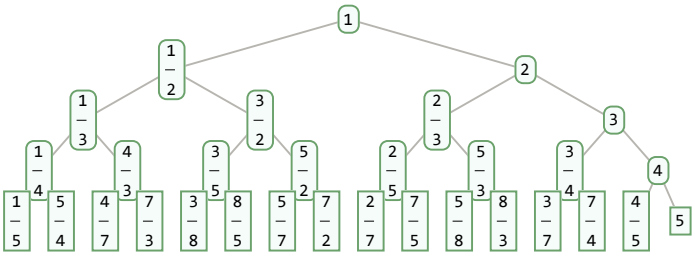
$$\left\{1, 2, 3, \frac{1}{2}, 5, 6, 7, 4, \frac{1}{3}, 10, 11, \frac{3}{2}, 13, 14, 15, \frac{1}{4}, 17, \frac{2}{3}, 19, \frac{5}{2}, 21, 22, 23, 12, \frac{1}{5}, 26, 9, \frac{7}{2}, 29, 30, 31, 8, 33, 34, 35, \frac{1}{6}, 37, 38, 39, 20, 41, 42, 43, \frac{11}{2}, \frac{5}{3}, 46, 47, \frac{3}{4}, \frac{1}{7}, \frac{2}{5}\right\}$$

The ordering function is well-defined, 1-1 and onto, but for example, 2/5 is far later in the list than 5/2, appearing after the integers 47 and 17, respectively. This may not necessarily be appropriate for some applications.

Calkin-Wilf Tree

In[•]:=
```
CalkinWilfTree[4]
```

Out[•]=



https://demonstrations.wolfram.com/CalkinWilfTreeOfFractions/

https://demonstrations.wolfram.com/TheTreeOfAllFractions/

The Calkin–Wilf binary tree has nodes of type $\frac{a}{b}$ (starting with $\frac{1}{1}$) with subnodes $\frac{a}{a+b}$ and $\frac{a+b}{b}$.

Note: this enumeration can be expressed in terms of the hyperbinary function.

In[53]:=
```
hb[0] = 1;
hb[n_?OddQ] := hb[n] = hb[(n - 1) / 2];
hb[n_?EvenQ] := hb[n] = With[{k = n / 2}, hb[k - 1] + hb[k]]
```

```
hb /@ Range[0, 25]
```

{1, 1, 2, 1, 3, 2, 3, 1, 4, 3, 5, 2, 5, 3, 4, 1, 5, 4, 7, 3, 8, 5, 7, 2, 7, 5}

(The additional "hb[n]=" in the recursion calls is *Mathematica*'s standard method of saving the results of all function calls so they will not need to be recalculated. Its effect can be seen by executing "?hb".)

```
? hb
```

Now the ordered list of rationals is obtained by forming ratios of adjacent elements of the hyperbinary list. Specifically,

In[56]:=
```
hbQ[n_] := hb[n - 1] / hb[n]
```

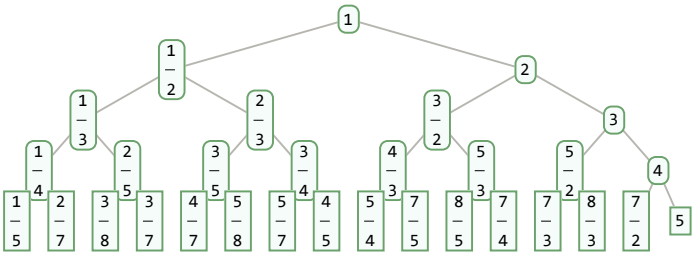
```
hbQ /@ Range[25]
```

$$\left\{1, \frac{1}{2}, 2, \frac{1}{3}, \frac{3}{2}, \frac{2}{3}, 3, \frac{1}{4}, \frac{4}{3}, \frac{3}{5}, \frac{5}{2}, \frac{2}{5}, \frac{5}{3}, \frac{3}{4}, 4, \frac{1}{5}, \frac{5}{4}, \frac{4}{7}, \frac{7}{3}, \frac{3}{8}, \frac{8}{5}, \frac{5}{7}, \frac{7}{2}, \frac{2}{7}, \frac{7}{5}\right\}$$

Stern-Brocot Tree

In[•]:=
```
SternBrocotTree[4]
```

Out[•]=



Note: A bit-reversal permutation of the Calkin–Wilf tree gives the Stern–Brocot tree.

https://demonstrations.wolfram.com/SternBrocotTree/

https://demonstrations.wolfram.com/FractionTreeAndContinuedFractions/

https://demonstrations.wolfram.com/TwoEnumerationsOfTheRationals/

Side-trip: Enumerations of all PPTs, NPTS, PTS (primitive / nearly-primitive Pythagorean Triples):

https://demonstrations.wolfram.com/PrimitivePythagoreanTriples3OrderedTreeGraph/

https://demonstrations.wolfram.com/ABinaryBasedEnumerationOfThePrimitivePythagoreanTriples/

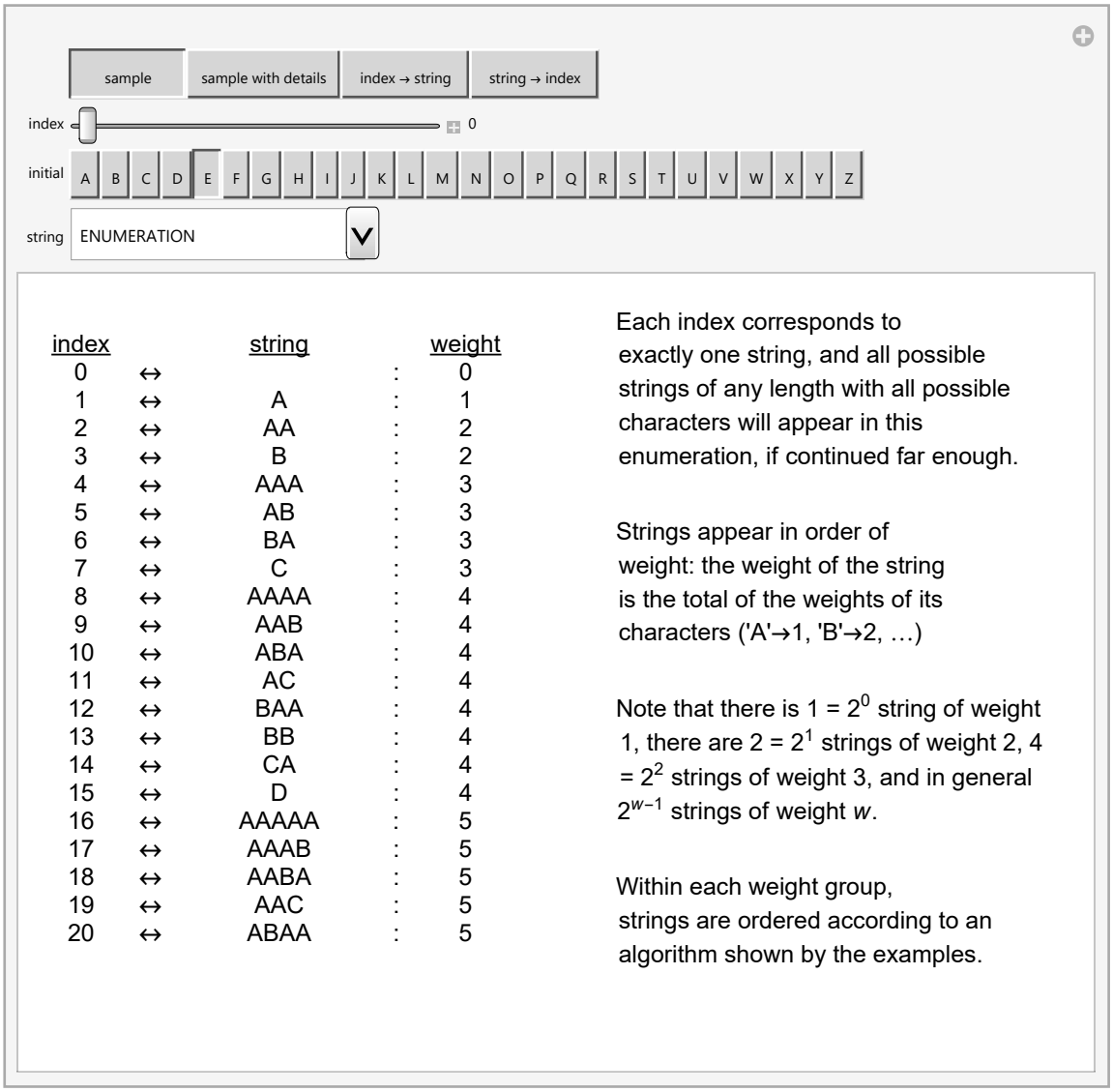
https://demonstrations.wolfram.com/EnumeratingPrimitivePythagoreanTriples/

https://demonstrations.wolfram.com/EnumerationOfAllPythagoreanTriples/

Enumeration of all Strings

The enumeration of all possible strings, unbounded in both length and size of alphabet used, was published in 2011, clarified as a binary tree in 2012:

https://demonstrations.wolfram.com/UniversalStringEnumeration/ :



index		string		weight
0	↔		:	0
1	↔	A	:	1
2	↔	AA	:	2
3	↔	B	:	2
4	↔	AAA	:	3
5	↔	AB	:	3
6	↔	BA	:	3
7	↔	C	:	3
8	↔	AAAA	:	4
9	↔	AAB	:	4
10	↔	ABA	:	4
11	↔	AC	:	4
12	↔	BAA	:	4
13	↔	BB	:	4
14	↔	CA	:	4
15	↔	D	:	4
16	↔	AAAAA	:	5
17	↔	AAAB	:	5
18	↔	AABA	:	5
19	↔	AAC	:	5
20	↔	ABAA	:	5

Each index corresponds to exactly one string, and all possible strings of any length with all possible characters will appear in this enumeration, if continued far enough.

Strings appear in order of weight: the weight of the string is the total of the weights of its characters ('A'→1, 'B'→2, …)

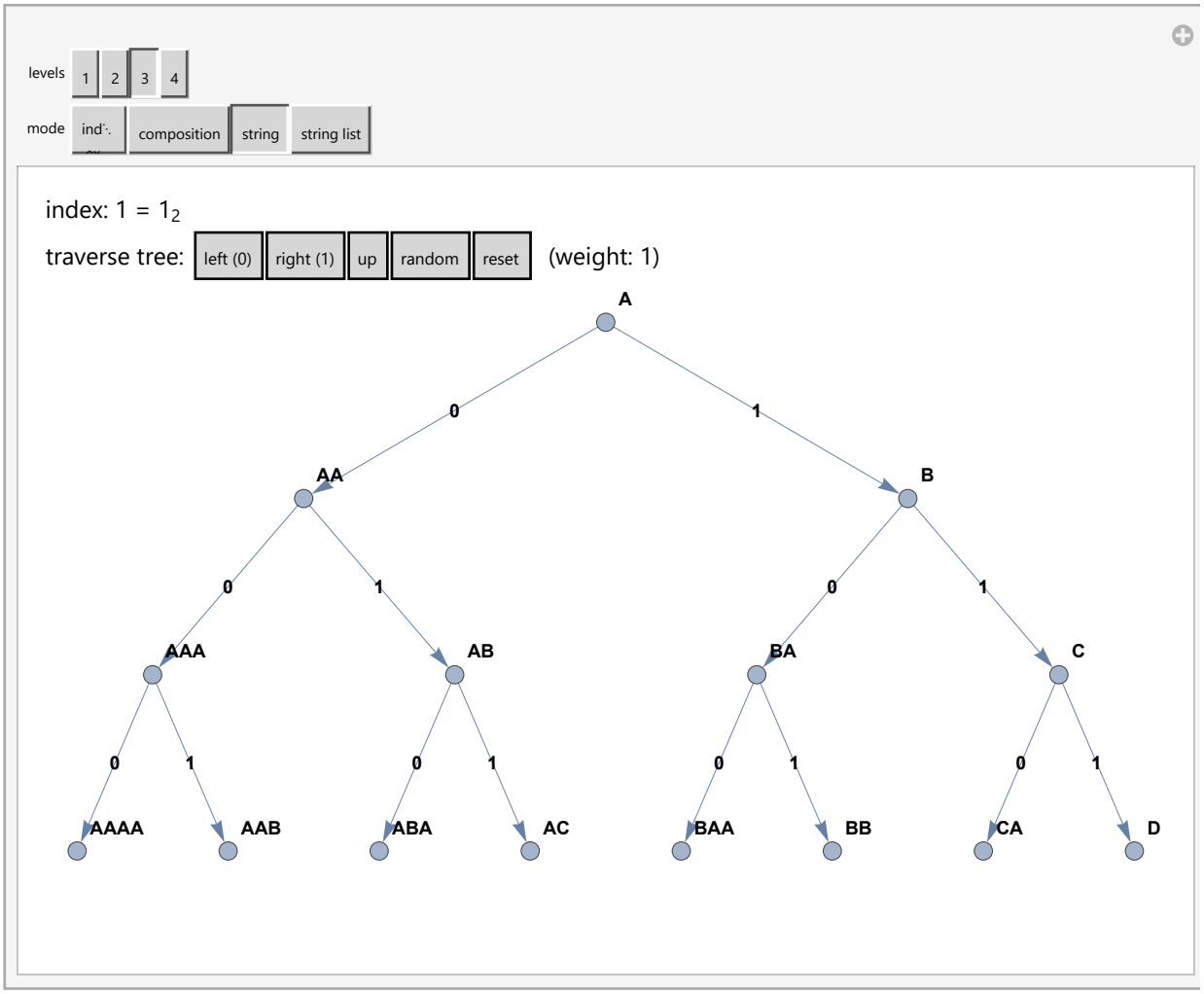
Note that there is $1 = 2^0$ string of weight 1, there are $2 = 2^1$ strings of weight 2, $4 = 2^2$ strings of weight 3, and in general 2^{w-1} strings of weight w.

Within each weight group, strings are ordered according to an algorithm shown by the examples.

In the enumeration, all strings appear in order of “weight”, with the weight originally maximally spread out. See “sample with details” to understand the algorithm.

https://demonstrations.wolfram.com/TreeOfStrings/ :

Out[69]=



This is the universal string enumeration shown in tree form. In the tree, descend to the Left in order to append an “A” to the string, to the Right to increment the final character of the string. The root of the tree is a single “A”. The digits shown on the branches leading to a particular node are the successive bits of the index corresponding to the string at the node. (First bit -- not shown -- is always a “1”.) So $11\,000_2$ corresponds to “BAAA”.

Original Sessie Enumeration

The first attempt to enumerate sessies was published here:

http://www.mathematica-journal.com/2011/05/indexing-strings-and-rulesets/

The article delineates the step-by-step process involved in creating a limited enumeration of strings, then a universal enumeration of all strings (think “all words in all languages”), an enumeration of all lists of strings (think “all books in all languages”), and finally a first attempt at enumerating all sessie rule sets. The latter did include all sessie rulesets, but unfortunately produced 25% of them twice. It

has now been replaced by enumerations that avoid that duplication and that allow “long-jump” predictions based on simpler mathematical relationships.

Updated Sessie Enumerations

Generalized Sessie Enumeration

We simplified and generalized the enumeration algorithm (the subject of a paper by Caviness, Case, Morrow & Kratzke, published in 2020). The new fully general algorithm based works equally well for rulesets of sequential substitution systems (sessies) and multi-way substitution system (“messies”). The code specifying rule set construction starts with one base-2 digit followed by base-5 digits. The first bit specifies whether (0) to use an empty string or (1) start with an “A” (a non-empty string) in the first rule set. Further base-5 codes progressively modify the rules, until the desired rule set has been constructed, as will be explained below. This construction effectively defines a one-to-one and onto relation between the positive integers and the set of all conceivable rulesets.

Here are the first 62 rule sets in the Generalized Sequential Substitution System Enumeration:

```
1 : 0₂ ₅ :     {"" -> "A"}
2 : 1₂ ₅ :     {"A" -> ""}
3 : 0₂0₅ :     {"" -> "A", "" -> "", "A" -> ""}
4 : 0₂1₅ :     {"" -> "A", "" -> "A"}
5 : 0₂2₅ :     {"" -> "A", "A" -> ""}
6 : 0₂3₅ :     {"" -> "AA"}
7 : 0₂4₅ :     {"" -> "B"}
8 : 1₂0₅ :     {"A" -> "", "" -> "A"}
9 : 1₂1₅ :     {"A" -> "", "A" -> ""}
10 : 1₂2₅ :    {"A" -> "A"}
11 : 1₂3₅ :    {"AA" -> ""}
12 : 1₂4₅ :    {"B" -> ""}
13 : 0₂00₅ :   {"" -> "A", "" -> "", "A" -> "", "" -> "A"}
14 : 0₂01₅ :   {"" -> "A", "" -> "", "A" -> "", "A" -> ""}
15 : 0₂02₅ :   {"" -> "A", "" -> "", "A" -> "A"}
16 : 0₂03₅ :   {"" -> "A", "" -> "", "AA" -> ""}
17 : 0₂04₅ :   {"" -> "A", "" -> "", "B" -> ""}
18 : 0₂10₅ :   {"" -> "A", "" -> "A", "" -> "", "A" -> ""}
19 : 0₂11₅ :   {"" -> "A", "" -> "A", "" -> "A"}
20 : 0₂12₅ :   {"" -> "A", "" -> "A", "A" -> ""}
21 : 0₂13₅ :   {"" -> "A", "" -> "AA"}
22 : 0₂14₅ :   {"" -> "A", "" -> "B"}
23 : 0₂20₅ :   {"" -> "A", "A" -> "", "" -> "A"}
24 : 0₂21₅ :   {"" -> "A", "A" -> "", "A" -> ""}
25 : 0₂22₅ :   {"" -> "A", "A" -> "A"}
26 : 0₂23₅ :   {"" -> "A", "AA" -> ""}
27 : 0₂24₅ :   {"" -> "A", "B" -> ""}
28 : 0₂30₅ :   {"" -> "AA", "" -> "", "A" -> ""}
29 : 0₂31₅ :   {"" -> "AA", "" -> "A"}
30 : 0₂32₅ :   {"" -> "AA", "A" -> ""}
31 : 0₂33₅ :   {"" -> "AAA"}
32 : 0₂34₅ :   {"" -> "AB"}
33 : 0₂40₅ :   {"" -> "B", "" -> "", "A" -> ""}
```

```
35 : 0241₅ :  {"" -> "B", "A" -> ""}
36 : 0243₅ :  {"" -> "BA"}
37 : 0244₅ :  {"" -> "C"}
38 : 1200₅ :  {"A" -> "", "" -> "A", "" -> "", "A" -> ""}
39 : 1201₅ :  {"A" -> "", "" -> "A", "" -> "A"}
40 : 1202₅ :  {"A" -> "", "" -> "A", "A" -> ""}
41 : 1203₅ :  {"A" -> "", "" -> "AA"}
42 : 1204₅ :  {"A" -> "", "" -> "B"}
43 : 1210₅ :  {"A" -> "", "A" -> "", "" -> "A"}
44 : 1211₅ :  {"A" -> "", "A" -> "", "A" -> ""}
45 : 1212₅ :  {"A" -> "", "A" -> "A"}
46 : 1213₅ :  {"A" -> "", "AA" -> ""}
47 : 1214₅ :  {"A" -> "", "B" -> ""}
48 : 1220₅ :  {"A" -> "A", "" -> "", "A" -> ""}
49 : 1221₅ :  {"A" -> "A", "" -> "A"}
50 : 1222₅ :  {"A" -> "A", "A" -> ""}
51 : 1223₅ :  {"A" -> "AA"}
52 : 1224₅ :  {"A" -> "B"}
53 : 1230₅ :  {"AA" -> "", "" -> "A"}
54 : 1231₅ :  {"AA" -> "", "A" -> ""}
55 : 1232₅ :  {"AA" -> "A"}
56 : 1233₅ :  {"AAA" -> ""}
57 : 1234₅ :  {"AB" -> ""}
58 : 1240₅ :  {"B" -> "", "" -> "A"}
59 : 1241₅ :  {"B" -> "", "A" -> ""}
60 : 1242₅ :  {"B" -> "A"}
61 : 1243₅ :  {"BA" -> ""}
62 : 1244₅ :  {"C" -> ""}
```

Reduced Sessie Enumeration

The Reduced Sessie Enumeration uses only quinary (base-5) codes, meaning that you never start with an empty string in the first rule. *fromReducedRank[n]* gives the n^{th} ruleset in the reduced quinary enumeration. The first 40 rulesets are listed below. If we define the weight of a ruleset as the sum of the individual letters, with A=1, B=2, C=3, …, it is clear that in this list (as in the GSSS Enumeration), weight 1 rulesets (containing only 1 letter, an A) all appear before weight 2 rulesets (containing either 2 As or 1 B), and weight 2 rulesets come before those of weight 3 in the list, etc. This ordering of rulesets is what makes “long-jumps” and the accelerated looping through the enumeration possible.

Since all rulesets containing an insertion rule (“”->something) will be discarded by *TestForConflictingRules*, if the insertion rule is not the last rule, the only way an insertion rule can be the first one is if the ruleset has only the one rule. A solo insertion rule can never delete cells, and so cannot produce a network. Therefore it is logical to eliminate all initial insertion rule cases, and this can easily be done by an adjustment to the enumeration itself, basically dropping the first (binary) digit of the Generalized Quinary enumeration.

```
In[•]:= {#⟦"Index"⟧, ": ", Subscript[ #⟦"QCode"⟧, 5], ": ", InputForm[#⟦"RuleSet"⟧]} & /@
          FromReducedRankIndex /@ Range[40] // Grid[#, Alignment → Left, Spacings → 0] &
```

```
Out[•]=
1 : 5    : {"A" → ""}
2 : 05   : {"A" → "", "" → "A"}
3 : 15   : {"A" → "", "A" → ""}
4 : 25   : {"A" → "A"}
5 : 35   : {"AA" → ""}
6 : 45   : {"B" → ""}
7 : 005  : {"A" → "", "" → "A", "" → "", "A" → ""}
8 : 015  : {"A" → "", "" → "A", "" → "A"}
9 : 025  : {"A" → "", "" → "A", "A" → ""}
10: 035  : {"A" → "", "" → "AA"}
11: 045  : {"A" → "", "" → "B"}
12: 105  : {"A" → "", "A" → "", "" → "A"}
13: 115  : {"A" → "", "A" → "", "A" → ""}
14: 125  : {"A" → "", "A" → "A"}
15: 135  : {"A" → "", "AA" → ""}
16: 145  : {"A" → "", "B" → ""}
17: 205  : {"A" → "A", "" → "", "A" → ""}
18: 215  : {"A" → "A", "" → "A"}
19: 225  : {"A" → "A", "A" → ""}
20: 235  : {"A" → "AA"}
21: 245  : {"A" → "B"}
22: 305  : {"AA" → "", "" → "A"}
23: 315  : {"AA" → "", "A" → ""}
24: 325  : {"AA" → "A"}
25: 335  : {"AAA" → ""}
26: 345  : {"AB" → ""}
27: 405  : {"B" → "", "" → "A"}
28: 415  : {"B" → "", "A" → ""}
29: 425  : {"B" → "A"}
30: 435  : {"BA" → ""}
31: 445  : {"C" → ""}
32: 0005: {"A" → "", "" → "A", "" → "", "A" → "", "" → "A"}
33: 0015: {"A" → "", "" → "A", "" → "", "A" → "", "A" → ""}
34: 0025: {"A" → "", "" → "A", "" → "", "A" → "A"}
35: 0035: {"A" → "", "" → "A", "" → "", "AA" → ""}
36: 0045: {"A" → "", "" → "A", "" → "", "B" → ""}
37: 0105: {"A" → "", "" → "A", "" → "A", "" → "", "A" → ""}
38: 0115: {"A" → "", "" → "A", "" → "A", "" → "A"}
39: 0125: {"A" → "", "" → "A", "" → "A", "A" → ""}
40: 0135: {"A" → "", "" → "A", "" → "AA"}
```

The RSS enumeration contains the same rulesets as the GSS enumeration in the same order, with initial insertion rules removed. The encoding is also the same, except for the removal of the initial bit that distinguished between rulesets beginning with an empty string and those not. The decision was made to use the RSS enumeration by default, since initial insertion rules have simple, well-understood behavior.

Within any given weight, advancing through the enumeration tends to compact the weight into fewer characters, moving it gradually towards the left. So the most spread out ruleset of weight 3, {“A”->“”,

“”->“A”,“”->“”,“A”->“”} is the first of this weight in the list, {“A”->“”,“”->“B”} appears further down, and {“C”->“”} is the last of this weight, with the entire weight compacted to the left.

Note: The “most spread out” rulesets have exactly 2 empty strings between single “A” strings, since we want to include the possibility of deletion rules (where the replacement string is “”) and insertion rules (where the search string is “”), and the possibility that an insertion might follow right after a deletion, thus putting 2 empty strings in sequence. E.g., {“A”->“”,“”->“B”}. But we will never need more than 2 consecutive empty strings! Unfortunately this algorithm also produces “”->“” rules, which are always unnecessary, as are non-final insertion rules. More on the solution to these problems below.

The algorithm is guaranteed to produce all valid rulesets somewhere in the list, if one goes far enough. It can easily be seen that there is no limit on the number of rules in the ruleset: all cases will eventually occur, and all combinations of all letters will eventually appear everywhere in the ruleset.

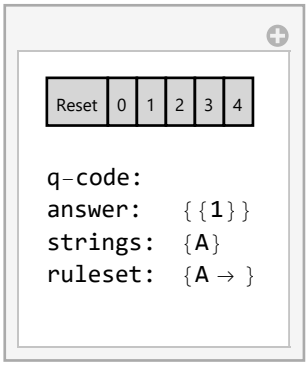
As already mentioned, this does generate some unneeded cases, e.g., with empty strings at too many positions in the ruleset, such as *nothing->nothing* rules, or multiple *nothing->something* rules -- only the first one could ever be used, or indeed any case where an insertion rule appears before the final position in the ruleset. But the most important thing is that all cases we want to study are included, and there is no repetition at all in the enumeration. It is also worth noting that this enumeration never produces invalid rulesets either.

Explanation of Q-Codes

- 0: end this string, insert two empty strings and start a new string with an "A"
- 1: end this string, insert one empty string and start a new string with an "A"
- 2: end this string and start a new string with an "A"
- 3: end this character and start a new character (as an "A")
- 4: increment this character

Try constructing some rulesets by hand, using the RSSS enumeration:

Out[70]=



Ruleset tests:

To save time and accelerate the use of the enumeration in an actual program, any ruleset that contains a problem can be skipped without generating the sessie and its network. In addition, the pattern and order of the enumeration allow us to skip over multiple cases at once. If such a "long-jump" is possible, the test returns the first ruleset in the enumeration that does *not* have that particular problem, i.e., the *target* of the long-jump. The main tests are:

Tests that eliminate the ruleset, and allow long-jumps:

Out[•]=

Test	Description
TestForConflictingRules	(* Two rules conflict if the second rule never has a chance to be used because the first rule preempts it. Ex/ {"C"→"", "A"→"B", "AA"→"C"}: rule 3 conflicts with rule 2, so the ruleset is equivalent to {"C"→"", "A"→"B"}, omitting rule 3 *)
TestForNonSoloIdentityRule	(* an identity rule is a rule in which a string is replaced by itself, no further rules will be used if the identity rule ever matches *)
(or TestForIdentityRule)	(* if solo identity rules are treated separately *)
TestForNonSoloInitialSubstringRule	(* Ex/ {"A"→"BAB", …} If 1st rule matches, it will never fail again. *)
TestForRenamedRuleSet	(* If the characters used in the strings of the ruleset are permuted or replaced by other characters, the SSS will look the same up to a permutation of colors, and the causal network will be identical. Ex/ {"BAB"→"ABA", "A"→"B", "B"→"AA"} is indistinguishable from {"ABA"→"BAB", "B"→"A", "A"→"BB"} *)
TestForUnusedRules	(* Still under construction: the idea is that if a rule is never used, it could be omitted without affecting the results. The difficulty is in being sure that the rule will never be used, rather than only rarely. *)

Tests that eliminate the ruleset, but don't allow long-jumps:

Out[•]=

Test	Description
TestForShorteningRuleSet	(* at least one rule shortens the state string, and none lengthen it, so this will eventually die out *)
TestForUnbalancedRuleSet	(* some letter appears on only one side of the rules, so that letter is either created or destroyed, but not both: that letter is not needed for the main pattern of the sessie *)

(*mod*) TestForAll checks all of these, in optimum order to allow longest possible long jumps.

Camille continued Christen's effort to identify the long jump target ruleset directly based on the quinary (base-5) index of the problem ruleset. The plan was to speed up the tests, as well as to increase understanding of the enumeration. This effort will now be transferred to the new base-3 algorithm (see below).

New Trinary Enumeration

This may (perhaps) supersede the quinary enumerations, since the math is significantly simpler, and the only disadvantage is the appearance of some invalid codes, which are easily jumped over, much as with any long jumps.

In the course of lengthy discussion, we have fine-tuned the new, trinary enumeration that will generate all sessie rulesets. The base-3 codes represent the following operations:

Out[•]=

0 = Increment last character (if there is one)
1 = Append an A
2 = Start a new empty/null string

A couple of points to note:

1. The math is simpler than using the original sessie enumeration (already published) or the generalized or reduced quinary (base-5) enumerations (published in CCMK). But unlike them, some forbidden codes exist, yielding no ruleset.

2. An initial "0" or a "20" anywhere in the trinary code is invalid, and should simply be skipped over in running through the enumeration. This is not a hardship, since it's easy to check and go on. The simpler math may make it worth it.

3. Our working version is to also drop any ruleset generated that has an odd number of strings. This can easily be checked by counting the number of times "2" appears in the trinary operation code, no need to even generate the ruleset! (Sadly, I see no way to long-jump over runs of such cases in the enumeration, so we just check and go on.)

4. The next task is to determine the easiest way to perform long-jumps over runs of invalid cases, non-final creation rulesets, conflicting rules, initial substring rules, etc. These are all situations that occur

in runs in the enumeration, so if found, the odds are we can long-jump over a run of similar cases, and so quickly move on to more interesting cases. (That is the strength of the enumeration.)

Below is the list of the first 50 positive integers, first in base 10, then in base 3, with the ruleset generated by the new enumeration algorithm, if any, shown. In case of an invalid ruleset, the "ok?" column indicates why: "…20…" means that the trinary code contains the digits 2 and 0 sequentially (new empty string followed by increment last character of current string: impossible), "odd" means that an odd number of strings was generated, a situation that does not correspond to a ruleset, which must have an integer number of rules, an even number of strings (each search string has a corresponding replacement string).

Out[•]=

Index	Base–3	Ok?	Ruleset
1	1	odd	
2	2		"" –> ""
3	10	odd	
4	11	odd	
5	12		"A" –> ""
6	20	...20...	
7	21		""–>"A"
8	22	odd	
9	100	odd	
10	101	odd	
11	102		"B"–>""
12	110	odd	
13	111	odd	
14	112		"AA"–>""
15	120	...20...	
16	121		"A"–>"A"
17	122	odd	
18	200	...20...	
19	201	...20...	
20	202	odd	
21	210		""–>"B"
22	211		""–>"AA"
23	212	odd	
24	220	odd	
25	221	odd	
26	222		""–>"","–>""
27	1000	odd	

28	1001	odd	
29	1002		"C"–>""
30	1010	odd	
31	1011	odd	
32	1012	...20...	
33	1020	...20...	
34	1021		"B"–>"A"
35	1022	odd	
36	1100	odd	
37	1101	odd	
38	1102		"AB"–>""
39	1110	odd	
40	1111	odd	
41	1112		"AAA"–>""
42	1120	...20...	
43	1121		"AA"–>"A"
44	1122	odd	
45	1200	...20...	
46	1201	...20...	
47	1202	odd	
48	1210		"A"–>"B"
49	1211		"A"–>"AA"
50	1212	odd	

Note that the presence of "…20…" anywhere in the trinary operation code means we can immediately increment the 0 to a 1, and zero out anything that follows: "… 20 …" => "… 21 000…0", rather than just incrementing the final digit of the code. This is a long-jump.

In the table below, all invalid cases have been omitted:

Out[•]=

Index	Base–3	Ok?	Ruleset
2	2		"" –> ""
5	12		"A" –> ""
7	21		""–>"A"
11	102		"B"–>""
14	112		"AA"–>""
16	121		"A"–>"A"
21	210		""–>"B"
22	211		""–>"AA"
26	222		""–>"","–>""
29	1002		"C"–>""
32	1012		"BA"–>""
34	1021		"B"–>"A"
38	1102		"AB"–>""
41	1112		"AAA"–>""
43	1121		"AA"–>"A"
48	1210		"A"–>"B"
49	1211		"A"–>"AA"
53	1222		"A"–>"","–>""
63	2100		""–>"C"
64	2101		""–>"BA"
66	2110		""–>"AB"
67	2111		""–>"AAA"
71	2122		""–>"A","–>""
77	2212		""–>"","A"–>""
79	2221		""–>"","–>"A"
83	10002		"D"–>""
86	10012		"CA"–>""
88	10021		"C"–>"A"
92	10102		"BB"–>""
95	10112		"BAA"–>""
97	10121		"BA"–>"A"

Next job: Identify what long-jumps can be triggered by the various ruleset tests, and find the simplest way to implement them. (Directly operating on the trinary code?)

Side-trip: Enumerations of Turing Machines:

https://demonstrations.wolfram.com/SmallTuringMachinesWithHaltingStateEnumerationAndRunnin

gOnAB/

https://demonstrations.wolfram.com/SimulationOfATuringMachine/