

# FAQ : SSS Enumeration

```
In[5]:= SetDirectory[ParentDirectory[NotebookDirectory[]]];
Needs["SSSiCv100`"];
```

## 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 It has now been replaced by enumerations that avoid that duplication and that allow “long-jump” predictions based on simpler mathematical relationships.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 : 025 : { "" -> "A" }
2 : 125 : { "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" }
```

```

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" -> "" }
34 : 02415 : { "" -> "B", "" -> "A" }
35 : 02425 : { "" -> "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  $n^{\text{th}}$  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 *TestForConflict-ingRules*, 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.

```

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"}

```

Out[*n*]=

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", ""→>"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

**\*\*\*Look at the RSSEnumerationIntro.pptx in Intro-Documents in 0intro to make animation\*\*\***

## 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[7]=

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[8]=

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).

## Updated Sessie Enumerations

This may 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[9]=

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).

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" -> ""

Out[15]=

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:

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"

Out[16]=

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?)