Indexing Strings and Rulesets

An Exploration Leading to an Enumeration

Kenneth E. Caviness

An enumeration of strings is developed, in which all strings of finite length of symbols from any alphabet appear, with no upper bounds for string length or alphabet size. A bijective indexing function and its inverse are found for the string enumeration, allowing iteration through the set of all strings, as well as identification of arbitrary strings by the associated index. The method is then extended to sequences of strings and to sequential substitution system (SSS) rulesets, providing a well-defined, relatively dense enumeration of all possible valid SSS rulesets for strings of arbitrary length and any number of symbols used in rulesets of any length, although in this case the indexing function is not one-to-one.

■ 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 [1]. Given a function `Enumeration` that returns the elements of a set in specified order, the following command finds the index of the first element that passes `TestFunction`. (The next cell does not evaluate.)

```
Catch[Do[If[TestFunction[Enumeration[i]], Throw[i]],
  {i, 1, SampleSize}]; None]
```

Sequential substitution systems 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. In this article enumeration systems are presented for all strings, for all lists of strings, and for all sequential substitution system rulesets. These enumerations can be used or modified for other applications based on rulesets and initial state strings (for example, nonsequential substitution systems, multiway systems, etc.).

■ Toward an Exhaustive List of Strings

□ Strings of Fixed Length and Fixed Number of Symbols

To generate a list of all possible strings of length three and made up of the characters A and B, the obvious method would be to think of a binary odometer having three positions, each able to display an A or a B—or a 0 or a 1. This is the normal increasing order of the first 2^3 binary numbers.

```
IntegerDigits[Range[0, 2^3 - 1], 2, 3]
```

```
{{0, 0, 0}, {0, 0, 1}, {0, 1, 0}, {0, 1, 1},
 {1, 0, 0}, {1, 0, 1}, {1, 1, 0}, {1, 1, 1}}
```

```
StringJoin[# /. {0 → "A", 1 → "B"}] & /@ %
```

```
{AAA, AAB, ABA, ABB, BAA, BAB, BBA, BBB}
```

We can identify each two-symbol length-three string with the corresponding numbers in the set {0, 1, …, 7}, providing an index into the sequence of strings, which appear in alphabetical order. But suppose we want to include *all* two-symbol strings, no matter what length? One possible ordering is the following.

```
{, A, B, AA, AB, BA, BB, AAA, AAB, ABA, ABB, BAA, BAB,
 BBA, BBB, AAAA, AAAB, AABA, AABB, ABAA, ABAB, ABBA,
 ABBB, BAAA, BAAB, BABA, BABB, BBAA, BBAB, BBBA, BBBB}
```

The pattern is: list all length-zero strings, then all length-one strings, then all length-two strings, then all length-three strings, etc. For a given length, go through the strings as you would odometer readings, changing the rightmost character most frequently, and others when the character to the right “rolls over.” Notice that the ordering is not alphabetic; if it were it would start with {"", "A", "AA", "AAA", …} and would never get to any strings that include B. But the order is well defined, with strings sorted first by length, then alphabetically within each string length group.

This method can be applied for any specified alphabet size. Below are functions to work with arbitrary-length strings with alphabet size b, here limited to 26 for the convenience of using the English alphabet, although the maximum for the function `IntegerString` is 36. (See [2] for an example of how to construct basically the same enumeration with no size limitation on the specified alphabet size by avoiding the use of an actual alphabet, listing characters by number only.)

```
RankStringInBase[b_Integer, s_String] :=
 Module[{len = StringLength[s], chars, digits},
   chars = CharacterRange["A", "Z"][[ ;; b]];
   digits = Join[CharacterRange["0", "9"],
      CharacterRange["A", "P"]][[ ;; b]];
   (b^len - 1) / (b - 1) +
    FromDigits[StringReplace[s, Thread[chars → digits]],
     b]] /; 2 ≤ b ≤ 26
RankStringInBase[1, s_String] :=
 StringLength[s] /; s == "" || Union[Characters[s]] == {"A"}

UnrankStringInBase[b_Integer, n_Integer] :=
 Module[{len, chars, digits},
   chars = CharacterRange["A", "Z"]〚 ;; b〛;
   digits =
    Join[CharacterRange["0", "9"],
      CharacterRange["A", "P"]]〚 ;; b〛;
   len = With[{$MaxExtraPrecision = 100},
     Floor@FullSimplify[Log[b, (n + 1) (b - 1)]]];
   StringReplace[
    ToUpperCase@IntegerString[n - (b^len - 1) / (b - 1), b, len] ,
    Thread[digits → chars]]] /; 2 ≤ b ≤ 26
UnrankStringInBase[1, n_Integer] :=
 StringJoin[Table["A", {n}]]
```

The single-symbol case is treated separately, simply counting or combining the appropriate number of characters. The more general formulas use as the string index the sum of the number of possible strings of length shorter than `len` and the base-b representation of the desired length string, where b is the alphabet size. Since for b symbols there are b^1 strings of length 1, b^2 strings of length 2, etc., the number of strings of length less than `len` is as follows.

```
Sum[b^k, {k, 0, len - 1}] // TraditionalForm
```

$$\frac{b^{\text{len}} - 1}{b - 1}$$

`UnrankStringInBase[b, n]` generates the string associated with the index number n in the listing of b-symbol strings, starting with 0. For $b = 2$, string length less than five, the indices can run from 0 to $\left(2^5 - 1\right) / (2 - 1) - 1$.

```
(2^5 - 1) / (2 - 1)
```

```
31
```

```
UnrankStringInBase[2, #] & /@ Range[0, 30]
```

```
{, A, B, AA, AB, BA, BB, AAA, AAB, ABA, ABB, BAA, BAB,
 BBA, BBB, AAAA, AAAB, AABA, AABB, ABAA, ABAB, ABBA,
 ABBB, BAAA, BAAB, BABA, BABB, BBAA, BBAB, BBBA, BBBB}
```

These are all strings of length less than five on an alphabet of two symbols. We use `RankStringInBase[b, s]` to return the index of each string.

```
RankStringInBase[2, #] & /@ %
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}
```

`RankStringInBase[b, s]` and `UnrankStringInBase[b, n]` are constructed to act as inverse functions, converting a string s into an integer n and vice versa, subject to the present limitations that $1 \leq b \leq 26$ and s be a string of uppercase characters taken from the first b letters of the English alphabet. `UnrankStringInBase[b, n]` produces a unique string s for each non-negative integer n, and `RankStringInBase[b, s]` reconstructs the index from the string. (Note that there is no limitation on string length or the index.)

Choose any string and an allowed (sufficiently large) base b.

```
RankStringInBase[26, "ABCDEFGHIJKLMNOPQRSTUVWXYZ"]
```

```
256 094 574 536 617 744 129 141 650 397 448 476
```

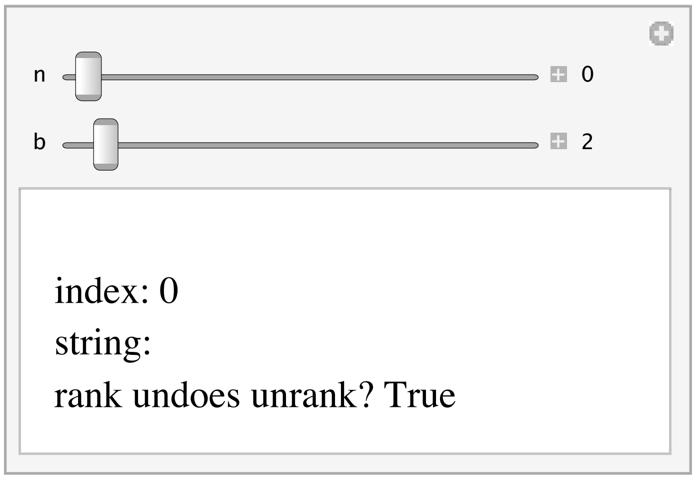
The inverse function retrieves the string.

```
UnrankStringInBase[26, %]
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Starting from an index is easier. We can iterate through its values, for example by moving a slider.

```
Manipulate[With[{s = UnrankStringInBase[b, n]},
  Text@
   Style[Column[{"\nindex: " <> ToString@n, "string: " <> s,
      "rank undoes unrank? " <>
       ToString[RankStringInBase[b, s] == n]}], 16]],
 {n, 0, 1000, 1, Appearance → "Labeled"},
 {{b, 2}, 1, 26, 1, Appearance → "Labeled"},
 SaveDefinitions → True]
```



```
index: 0
string:
rank undoes unrank? True
```

Note that `UnrankStringInBase` could even be used in an infinite loop, iterating through all strings for a given set of symbols, listing shorter strings before longer ones. In the following loop, rather than using `While[True, ]`, we stop when the string length exceeds three.

```
n = 0; slist = {};
While[(s = UnrankStringInBase[5, n]; StringLength[s] ≤ 3),
  AppendTo[slist, s]; n++];
Clear[n, s];
slist
```

```
{, A, B, C, D, E, AA, AB, AC, AD, AE, BA, BB, BC, BD, BE, CA, CB,
 CC, CD, CE, DA, DB, DC, DD, DE, EA, EB, EC, ED, EE, AAA,
 AAB, AAC, AAD, AAE, ABA, ABB, ABC, ABD, ABE, ACA, ACB, ACC,
 ACD, ACE, ADA, ADB, ADC, ADD, ADE, AEA, AEB, AEC, AED, AEE,
 BAA, BAB, BAC, BAD, BAE, BBA, BBB, BBC, BBD, BBE, BCA, BCB,
 BCC, BCD, BCE, BDA, BDB, BDC, BDD, BDE, BEA, BEB, BEC,
 BED, BEE, CAA, CAB, CAC, CAD, CAE, CBA, CBB, CBC, CBD,
 CBE, CCA, CCB, CCC, CCD, CCE, CDA, CDB, CDC, CDD, CDE,
 CEA, CEB, CEC, CED, CEE, DAA, DAB, DAC, DAD, DAE, DBA,
 DBB, DBC, DBD, DBE, DCA, DCB, DCC, DCD, DCE, DDA, DDB,
 DDC, DDD, DDE, DEA, DEB, DEC, DED, DEE, EAA, EAB, EAC,
 EAD, EAE, EBA, EBB, EBC, EBD, EBE, ECA, ECB, ECC, ECD,
 ECE, EDA, EDB, EDC, EDD, EDE, EEA, EEB, EEC, EED, EEE}
```

There are several nice things about these functions. `UnrankStringInBase` enumerates the b-symbol strings in the specified order (sorted with shorter strings first, alphabetically within each string length), giving a "standard order" for listing these strings without omissions. But notice that `RankStringInBase` and `UnrankStringInBase` do not generate this (infinite!) list and look for a match or pick out an element, rather they build the string from the index or deduce the index from the string, respectively.

□ Duplication of Effort

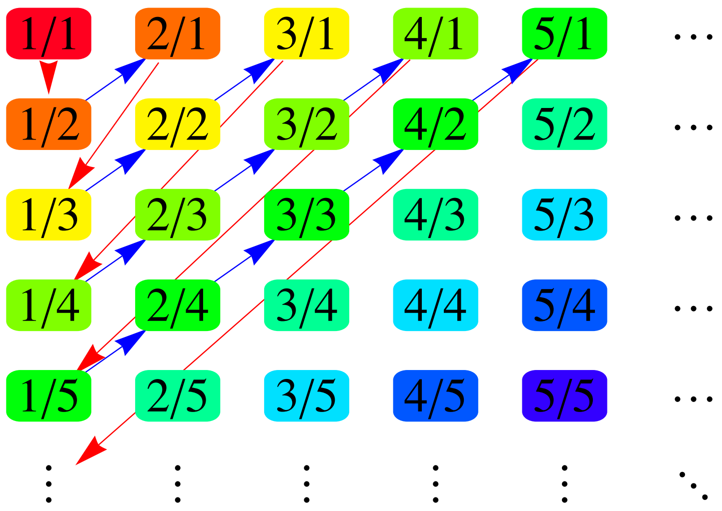
Of course, the particular enumeration order chosen motivates the creation of the rank and unrank functions. The above method includes all possible strings written using an alphabet of b symbols, and does so in an explainable order. But although by construction it allows all string lengths, whenever the alphabet size is increased all the previous work must be redone.

Is there a way to list all strings, allowing both string length and alphabet size to grow without upper bound? (For the sake of argument let us assume that the symbols themselves can be written down in some order, perhaps Unicode order followed by the order in which new symbols are invented.) What is needed is some way to add to the list without rearranging or repeating previously listed entries. For inspiration we turn to a similar situation, the enumeration of the rational numbers.

■ Rational Examples

□ 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 (see Figure 1, below).



▲ **Figure 1**. Cantor's diagonal ordering of the rationals: coloration added to highlight diagonal rows.

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 $\{1/1, 2/1, 3/1, \ldots 1/2, 2/2, 3/2, \ldots, 1/3, 2/3, 3/3, \ldots\}$, this method allows growth in both directions to continue indefinitely, following a defined pattern while clearly including all possible combinations. (See [3] for an alternative route through the array.) Of course, one drawback of this method as applied to fractions is that equivalent fractions get counted multiple times. For example $1/1 = 2/2 = 3/3 = \ldots$, $1/2 = 2/4 = 3/6 = \ldots$, etc. But the mathematical literature contains many examples of nonrepetitive ways of ordering the rationals (e.g., [4, 5, 6, 7, 8]).

None of the nonrepetitive 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 will consider nonrepetition a desirable but not necessary feature.

What is essential then?

1. The list should be unambiguous: it can be generated to any desired number of elements, and the order can be unambiguously described. Here the fractions are listed in increasing order first by the sum of numerator and denominator (as colored in Figure 1), next by numerator.

2. A successor algorithm should exist: from a given fraction n/d, can the next fraction in the list be found? Yes, if $d > 1$, the next fraction is $(n+1)/(d-1)$; 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. This can be implemented in this way as two-element lists.

```
Successor[{n_Integer, d_Integer}] :=
 If[GCD @@ # == 1, #, Successor@#] & @
  If[d == 1, {1, (n + 1)}, {(n + 1), (d - 1)}]
```

Here are the first 25 successors of `{1, 1}`, shown in fractional form.

```
FractionBox @@@ NestList[Successor, {1, 1}, 25] //
 DisplayForm
```

$$\left\{\frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{1}{3}, \frac{3}{1}, \frac{1}{4}, \frac{2}{3}, \frac{3}{2}, \frac{4}{1}, \frac{1}{5}, \frac{5}{1}, \frac{1}{6}, \frac{2}{5}, \frac{3}{4}, \frac{4}{3}, \frac{5}{2}, \frac{6}{1}, \frac{1}{7}, \frac{3}{5}, \frac{5}{3}, \frac{7}{1}, \frac{1}{8}, \frac{2}{7}, \frac{4}{5}, \frac{5}{4}, \frac{7}{2}\right\}$$

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 $(n+d-1)^{\text{th}}$ and is element n on that diagonal. An easy way to do this is to create a function to generate the n^{th} triangular number (the total number of entries in the previous diagonals).

```
Tri[n_] := Evaluate[∑_{k=1}^{n} k]; ? Tri
```

```
Global`Tri
```

```
Tri[n_] := 1/2 n (1 + n)
```

Now the ranking function is easy.

```
RankRational[r_] :=
 (Tri[# + Denominator[r] - 2] + #) & @ Numerator[r]
```

```
RankRational /@ {1 / 1, 1 / 2, 2 / 1, 1 / 3, 2 / 2, 3 / 1, 1 / 4,
  2 / 3, 3 / 2, 4 / 1}
```

```
{1, 2, 3, 4, 1, 6, 7, 8, 9, 10}
```

Except for unreduced fractions these are in ascending order, as desired. An unranking function will facilitate testing for unreduced fractions.

```
Solve[x == Tri[n], n]
```

$$\left\{\left\{n \to \frac{1}{2}\left(-1-\sqrt{1+8x}\right)\right\}, \left\{n \to \frac{1}{2}\left(-1+\sqrt{1+8x}\right)\right\}\right\}$$

invTri[x_Integer] := Ceiling$\left[\frac{1}{2}\left(-1+\sqrt{1+8x}\right)\right]$

```
UnrankRational[x_Integer] := Module[{k, n, d},
  k = invTri[x];
  n = x - Tri[k - 1];
  d = k - n + 1;
  If[GCD[n, d] > 1, 0, n / d]]
 (* returns 0 instead of unreduced fractions *)
```

```
UnrankRational /@ Range[25]
```

$$\left\{1, \frac{1}{2}, 2, \frac{1}{3}, 0, 3, \frac{1}{4}, \frac{2}{3}, \frac{3}{2}, 4, \frac{1}{5}, 0, 0, 0, 5, \frac{1}{6}, \frac{2}{5}, \frac{3}{4}, \frac{4}{3}, \frac{5}{2}, 6, \frac{1}{7}, 0, \frac{3}{5}, 0\right\}$$

```
RankRational /@ %
```

```
{1, 2, 3, 4, 0, 6, 7, 8, 9, 10, 11, 0, 0,
 0, 15, 16, 17, 18, 19, 20, 21, 22, 0, 24, 0}
```

Since this method of listing the rationals includes duplicates, the `UnrankRational` function returns 0 instead of a duplicate. However the nonzero fractions returned are unique and appear in the defined order.

Again, although it would be nice for the mapping or indexing method to be one to one, it is not necessary.

□ Nonrepetitive Indexing of the Rationals

As mentioned above, there are one to one 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 [4] 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 the order of the factors).

```
showFactorization =
  Row[{#, " = ", Row[Superscript @@@ FactorInteger[#],
      "×"]}] & ;
showFactorization[174 636 000]
```

```
174 636 000  =  2^5 × 3^4 × 5^3 × 7^2 × 11^1
```

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 $\{5, 4, 3, 2, 1, 0, 0, 0, \ldots\}$. 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 the denominator, a fact that motivates the following algorithm, in which odd exponents define factors of the numerator and even exponents define factors of the denominator.

```
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]
```

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

```
showFactorization /@ {2200, 63} // Column
```

```
2200  =  2^3 × 5^2 × 11^1
63  =  3^2 × 7^1
```

Note that even exponents are halved and then used as the exponents of the same prime factors in the denominator, odd exponents incremented, then halved and similarly used to specify the numerator. (For an extension to all rationals see [9].) Of course this procedure is not unique, the treatment of odd and even exponents 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, one to one, 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.

Another bijective ordering of the rationals, due to Calkin and Wilf [6], is treated here. This ordering can be expressed in terms of the hyperbinary function, which can be recursively defined as follows.

```
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 (called memoization) of saving the results of a function call so that it will not need to be recalculated after the first time. Its effect can be seen by executing `?hb`.)

```
? hb
```

```
Global`hb
```

```
hb[0] = 1

hb[1] = 1
```

```
hb[2] = 2

hb[3] = 1

hb[4] = 3

hb[5] = 2

hb[6] = 3

hb[7] = 1

hb[8] = 4

hb[9] = 3

hb[10] = 5

hb[11] = 2

hb[12] = 5

hb[13] = 3

hb[14] = 4

hb[15] = 1

hb[16] = 5

hb[17] = 4

hb[18] = 7

hb[19] = 3

hb[20] = 8

hb[21] = 5

hb[22] = 7

hb[23] = 2

hb[24] = 7

hb[25] = 5
```

hb[n_?OddQ] := hb[n] = hb$\left[\frac{n-1}{2}\right]$

```
hb[n_?EvenQ] := hb[n] = With[{k = n/2}, hb[k - 1] + hb[k]]
```

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

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

hbQ /@ Range[25]
```

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

Besides containing no duplicate or unreduced fractions, this ordering has the property that fractions with a small numerator and denominator tend to appear before those with larger ones. An inverse function can be created using the hyperbinary numbers as a look-up table, but musings in [10] motivate a more direct approach. Consider the numerator n and denominator d of the reduced fraction: we begin constructing a sequence of 0s and 1s by recording a 0 if $n < d$ or a 1 if $n > d$. Then a new fraction is formed by replacing the larger (of the numerator and denominator) by their difference, and repeat. Nice features of this process are:

1. Each fraction so produced is automatically in reduced form.
2. Either the numerator or the denominator of each fraction is smaller than that of the preceding one.
3. The process will inevitably terminate when $n = d = 1$.

Now this sequence of binary digits can be interpreted as a unique integer. To recover the index of the fraction, we use the digit sequence in the reverse of the order in which it was generated, and prepend a 1 to distinguish shorter digit sequences from sequences with initial 0s. (This corresponds to counting the number of possible shorter sequences and adding it to the index—a concept we return to when considering the indexing of the set of all strings.)

```
hbQInverse[r : (_Rational | _Integer)] :=
 Module[{n = Numerator[r], d = Denominator[r], seq = {}},
  While[n > 1 || d > 1, If[n < d, seq = {0, seq}; d = d - n,
    seq = {1, seq}; n = n - d]];
  FromDigits[Flatten@{1, seq}, 2]]
```

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

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

```
hbQInverse /@ %
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25}
```

Another excellent feature of this algorithm is that it lends itself to the creation of a successor function, such as was possible for the simple diagonal method of ordering fractions.

```
hbQSuccessor[r : (_Rational | _Integer)] := 1 / (1 + 2 Floor[r] - r)
```

```
NestList[hbQSuccessor, 1, 24]
```

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

■ One-to-One and Onto Indexing of All Strings?

□ First Attempt

How can we do something similar with strings of any length and any alphabet size? Suppose we lay out subsets of strings having m symbols and length n and use the diagonal method to choose which subset to include next in the set of strings of all lengths and all number of symbols.

$$\begin{pmatrix} \text{strings}[1,1] & \text{strings}[2,1] & \text{strings}[3,1] & \cdots \\ \text{strings}[1,2] & \text{strings}[2,2] & \text{strings}[3,2] & \cdots \\ \text{strings}[1,3] & \text{strings}[2,3] & \text{strings}[3,3] & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

In this scenario, `strings[2, 3]` is the subset of all strings of length three using an alphabet of two characters. Each subset is finite (`strings[m, n]` has m^n elements), and so the diagonal method ensures that we will eventually get to any given subset.

The situation is not analogous to the diagonal listing of fractions, since `strings[2, 2]` ≠ `strings[1, 1]`, etc. So there are no duplicate subsets to remove.

```
strings[b_Integer, strlen_Integer] := StringJoin /@
   (IntegerDigits[Range[0, b^strlen - 1], b, strlen] /.
     Thread[Range[0, b - 1] →
       CharacterRange["A", "Z"][[ ;; b]]]) /; b ≥ 2
strings[1, strlen_Integer] :=
  {StringJoin@Table["A", {strlen}]};
strings[0, 0] = {""};
InfiniteMatrixForm[a_] :=
  Module[{rows, cols, b, c, d}, {rows, cols} = Dimensions[a];
   b = Table[{"⋯"}, {rows}]; c = {Table["⋮", {cols}]};
   d = {{"⋱"}}; MatrixForm[ArrayFlatten[{{a, b}, {c, d}}]]];
InfiniteMatrixForm@
 Table[If[n + b ≤ 5, strings[b, n], "⋮"], {n, 1, 3}, {b, 1, 3}]
```

```
(  {A}                       {A, B}                                       {A, B, C}                      ⋯ )
(  {AA}                 {AA, AB, BA, BB}                  {AA, AB, AC, BA, BB, BC, CA, CB, CC}   ⋯ )
( {AAA}  {AAA, AAB, ABA, ABB, BAA, BAB, BBA, BBB}                      ⋮                         ⋯ )
(   ⋮                           ⋮                                           ⋮                        ⋱ )
```

For completeness we also defined `strings[0, 0]` as the set containing only the zero-length string, and will let this be the first element in our overall list of strings of any length and any number of characters. Now the list of strings of any length and number of characters, up to the fourth diagonal, is as follows.

```
stringsUpToDiagonal[d_Integer] := Module[{strlen, k, n},
   Flatten[{strings[0, 0],
     Table[Table[strings[b, n + 1 - b], {b, 1, n}],
      {n, 1, d}]}]];
stringsUpToDiagonal[4]
```

```
{, A, AA, A, B, AAA, AA, AB, BA, BB, A, B,
 C, AAAA, AAA, AAB, ABA, ABB, BAA, BAB, BBA, BBB,
 AA, AB, AC, BA, BB, BC, CA, CB, CC, A, B, C, D}
```

We see that there are indeed duplicate strings in our list. In fact, every string list in the array layout is a subset of its neighbor to the right: `strings[n, k]` ⊂ `strings[n, k + 1]`. Let us remove the duplicates in the sample above.

```
testlist = DeleteDuplicates[%]
```

```
{, A, AA, B, AAA, AB, BA, BB, C, AAAA, AAB, ABA,
 ABB, BAA, BAB, BBA, BBB, AC, BC, CA, CB, CC, D}
```

□ Rank and Unrank Strings of *n* Symbols?

We could define rank and unrank functions that simply skipped over these duplicates (as in the simple diagonalization ordering of fractions), but might there not be a way of listing all strings without duplicates? The list above can be thought of, to a first approximation, as a list sorted by weight, where the weight of a string is the sum of the weights of its characters and the weights of the characters increase in alphabetical order in some fashion. Suppose we try `"A" → 1, "B" → 2, "C" → 3, ….` For convenience we again stop at $b = 26$, but could just as easily continue through all symbols in some agreed upon order.

```
StringWeight[s_String] :=
 Total[Characters[s] /. characterWeights];
characterWeights =
 Prepend[Thread[CharacterRange["A", "Z"] → Range[26]], "" → 0]
```

```
{ → 0, A → 1, B → 2, C → 3, D → 4, E → 5, F → 6,
 G → 7, H → 8, I → 9, J → 10, K → 11, L → 12, M → 13,
 N → 14, O → 15, P → 16, Q → 17, R → 18, S → 19,
 T → 20, U → 21, V → 22, W → 23, X → 24, Y → 25, Z → 26}
```

```
Grid[Transpose@{testlist, StringWeight /@ testlist}]
```

```
      0
  A   1
  AA  2
  B   2
 AAA  3
  AB  3
  BA  3
  BB  4
  C   3
AAAA  4
 AAB  4
 ABA  4
 ABB  5
 BAA  4
 BAB  5
 BBA  5
 BBB  6
  AC  4
  BC  5
  CA  4
  CB  5
  CC  6
  D   4
```

The reason the string weights do not appear in nondecreasing order is because the way the list was formed does not follow from this simple weighting scheme. In order to create rank and unrank functions we might figure out some other way to index this list, or we could index the more easily understood list including duplicates, and just identify and then ignore duplicates. But suppose instead we *start* with a weighting scheme and generate a string from it?

The simple weighting scheme `"A"` → `1, "B"` → `2, "C"` → `3, …`, together with the definition of the weight of a string as the sum of the weights of its characters, suggests a string enumeration that lists strings in increasing order of weight, all strings of weight n appearing before those of weight $n+1$, for any non-negative integer n. For example, the strings with weights 1 to 4 are shown in this table.

```
1:   A
2:   AA     B
3:   AAA    AB    BA    C
4:   AAAA   AAB   ABA   BAA   BB   AC   CA   D
```

The enumeration could simply be the concatenation of these rows into a single list, with some additional ordering system for arranging the strings within the rows. Clearly any finite string s has some positive integer weight and will therefore appear somewhere in the table. It will also become clear that each row is finite, so s will appear in the enumeration at a position dependent on its position within its row (weight group) and the total number of strings of lesser weight. To clarify these statements and provide an algorithm for the enumeration, we turn to the concept of *integer compositions*.

□ Integer Compositions and Partitions

In number theory, partitions and compositions of a positive integer n are ways of writing n as a sum of positive integers [11, 12]. Here are the ways of dividing 4 into positive integer parts.

```
IntegerPartitions[4]
```

```
{{4}, {3, 1}, {2, 2}, {2, 1, 1}, {1, 1, 1, 1}}
```

That is to say, 4 can be written as the sum of positive integers in the following ways.

```
Row[#, "+"] & /@ IntegerPartitions[4]
```

```
{4, 3+1, 2+2, 2+1+1, 1+1+1+1}
```

In the case of an integer partition, the order is irrelevant. By contrast, permutations of addends may result in distinct integer compositions.

```
IntegerCompositions[n_Integer] :=
  Flatten[Permutations /@ IntegerPartitions[n], 1];
```

```
Row[#, "+"] & /@ IntegerCompositions[4]
```

```
{4, 3+1, 1+3, 2+2, 2+1+1, 1+2+1, 1+1+2, 1+1+1+1}
```

Surprisingly enough, the number of compositions of n is a simple formula. By inspection we get a first clue.

```
Text[{#, Length[IntegerCompositions[#]]} & /@ Range[7] //
   Transpose //
  TableForm[#, TableHeadings →
     {{Text@Style["n", Italic],
       Row[{"number of compositions of ",
         Style["n", Italic]}]}, None}] &]
```

n	1	2	3	4	5	6	7
number of compositions of *n*	1	2	4	8	16	32	64

Hypothesis: the number of integer compositions of n is 2^{n-1}.

This result suggests the existence of a natural method of associating binary numbers and integer compositions.

Any integer composition of a positive integer n (assuming nonzero parts) can be thought of as a sequence of 1s, interleaved by either commas or plus signs. Since there are $n-1$ positions where two choices can be made (that is, whether to insert **,** or $+$), the number of possible results is 2^{n-1}. For instance, the compositions of 4 shown above can all be represented as follows.

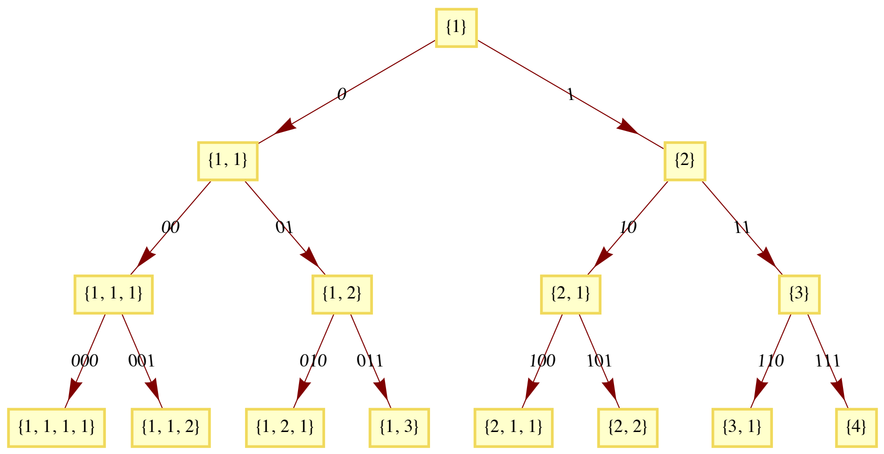
$$\left\{1 \underset{,}{\overset{+}{}} 1 \underset{,}{\overset{+}{}} 1 \underset{,}{\overset{+}{}} 1\right\}$$

An obvious ordering of the compositions is to let 0 and 1 represent a plus sign and a comma, respectively, so that each composition can be indexed by a binary number. For example, there are $2^{4-1} = 8$ compositions of 4.

0	=	000_2	$\Longleftrightarrow$	{1,1,1,1}	=	{1, 1, 1, 1}
1	=	001_2	$\Longleftrightarrow$	{1,1,1+1}	=	{1, 1, 2}
2	=	010_2	$\Longleftrightarrow$	{1,1+1,1}	=	{1, 2, 1}
3	=	011_2	$\Longleftrightarrow$	{1,1+1+1}	=	{1, 3}
4	=	100_2	$\Longleftrightarrow$	{1+1,1,1}	=	{2, 1, 1}
5	=	101_2	$\Longleftrightarrow$	{1+1,1+1}	=	{2, 2}
6	=	110_2	$\Longleftrightarrow$	{1+1+1,1}	=	{3, 1}
7	=	111_2	$\Longleftrightarrow$	{1+1+1+1}	=	{4}

The next output shows graphically how the bits of the index are used to determine which integer composition is intended. Each 0 in the binary code is interpreted as the instruction "insert a comma before the next 1 in the composition" or "end this integer, start the next as a 1"; each 1 is an instruction to "insert a plus sign before the next 1" or "increment the last integer in the composition."

This shows the beginning of the tree of all integer compositions, showing compositions of n on level n.

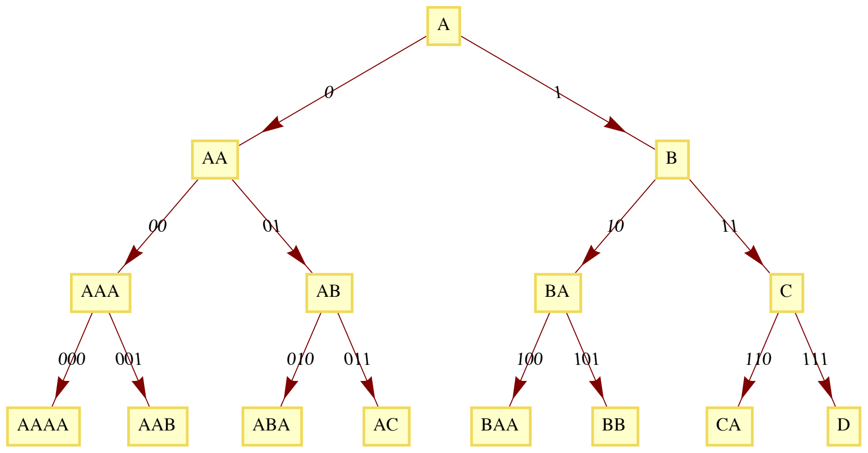


■ A New String Ordering

This elegant concept immediately gives us an unambiguous order for the set of all strings as well: we need only let $n = 0, 1, 2, \ldots$ and run through all 2^n possibilities for each n, in all cases letting the integer compositions obtained represent strings using the simple substitutions defined in `characterWeights`. The following function produces the nth string of weight w by treating the 0s and 1s of the $(w - 1)$-bit binary representation of n as instructions to either stop or continue incrementing the last digit of the string.

```
StringN[weight_Integer, n_Integer] := Module[{t = 1},
  Reap[Scan[If[# == 0, Sow[t]; t = 1, t++] &,
       IntegerDigits[n, 2, weight - 1]]; Sow[t]][[2, 1]]
    /. (Reverse /@ characterWeights) // StringJoin]
```

Here is the same tree as before, but with the integer compositions translated into strings. Note that all strings of weight n appear on level n of the tree. Each 0 means "append an `A` to the string"; each 1 means "increment the final character of the string."



Here then are all the strings of weight 4.

```
StringN[4, #] & /@ Range[0, 2^(4-1) - 1]
```

```
{AAAA, AAB, ABA, AC, BAA, BB, CA, D}
```

Notice that `StringN` is in fact an unrank function for strings of any specified weight, and at the same time it provides a way to generate the list of all such strings. And since each such list is finite, we can join lists of successively greater weighting values to create the universal list of all strings. (Recall the diagonal ordering of fractions first by the sum of numerator and denominator, then by numerator, in which each diagonal was finite.)

```
Text@Column@Table[StringN[w, #] & /@ Range[0, 2^(w-1) - 1],
   {w, 1, 5}]
```

```
{A}
{AA, B}
{AAA, AB, BA, C}
{AAAA, AAB, ABA, AC, BAA, BB, CA, D}
{AAAAA, AAAB, AABA, AAC, ABAA, ABB,
 ACA, AD, BAAA, BAB, BBA, BC, CAA, CB, DA, E}
```

Remember that level n consists of all 2^n strings (no matter what length) of weight n. In order to index this universal string list, we need to know how many strings there are in the levels up to and including n.

```
Sum[2^(k-1), {k, 1, n}] // TraditionalForm
```

$2^n - 1$

If we include the empty string of length zero, there are 2^n strings of weight less than or equal to n. So given an index i, we first find the largest value n such that $2^n < i$, then pass $i - 2^n$ as the index into the list of strings with weight $n + 1$, returning `StringN[n + 1, i - 2^n]`. (The next cell does not evaluate.)

```
UnrankString[i_Integer] :=
  With[{n = Floor[Log[2, i]]}, StringN[n + 1, i - 2^n]];
```

The above definition is included only for clarity. For increased computational speed we replace it by the following functionally equivalent version using `BitLength`, a function that returns the number of binary digits needed to express an integer: `BitLength[i] = Floor[Log[2, i]] + 1`. (The author is indebted to the reviewers of [13] for this suggestion.)

```
UnrankString[i_Integer] :=
  With[{n = BitLength[i]}, StringN[n, i - 2^(n - 1)]];
UnrankString[0] = "";
```

```
UnrankString /@ Range[0, 32]

{, A, AA, B, AAA, AB, BA, C, AAAA, AAB, ABA, AC,
 BAA, BB, CA, D, AAAAA, AAAB, AABA, AAC, ABAA, ABB,
 ACA, AD, BAAA, BAB, BBA, BC, CAA, CB, DA, E, AAAAAA}
```

The inverse algorithm goes like this: Note the total string weight w. Insert a 0 between all letters and then replace the letters `"A", "B"`, …, by strings of increasing numbers of 1s: `"A"` → `""`, `"B"` → `"1"`, `"C"` → `"11"`, …. Add the resulting binary number to 2^{w-1}, the number of strings of weight less than w. But wait, 2^{w-1} in binary is $1\,0\,0\overbrace{\cdots}^{w}0$, and all strings of weight w have been encoded as length w binary numbers, so we can add 2^{w-1} by simply prefixing `1` to the binary number found above. This is reminiscent of the Calkin–Wilf indexing of the rationals, where the index describes a path through the binary tree containing the (positive) rationals.

```
RankString[s_String] :=
  With[{weight = Characters[s] /. characterWeights},
   FromDigits[
    Flatten[{1, Riffle[Table[1, {#}] & /@ (weight - 1), 0]}],
    2]];
RankString[""] = 0;

UnrankString /@ Range[0, 32]

{, A, AA, B, AAA, AB, BA, C, AAAA, AAB, ABA, AC,
 BAA, BB, CA, D, AAAAA, AAAB, AABA, AAC, ABAA, ABB,
 ACA, AD, BAAA, BAB, BBA, BC, CAA, CB, DA, E, AAAAAA}

RankString /@ %

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32}
```

It appears that `RankString` functions correctly and gives back the index that generated each string. Here is a slightly longer test.

```
Range[0, 10000] ==
 RankString /@ UnrankString /@ Range[0, 10000]

True
```

Of course, no finite test can prove that `RankString` and `UnrankString` are indeed inverse functions: that claim is based on the unambiguity of the enumeration (strings appear in order of increasing weight, ordered within weight group by composition index). Theoretical considerations [14] indicate that the set of all (finite-length) words that can be formed from a countably infinite alphabet is countably infinite. The above enumeration of all strings is a one-to-one function from the set of all (finite-length) strings onto the set of positive integers and thus provides a direct demonstration that the set is countably infinite.

We use the trivial definition of `NextString` because it is simpler and almost as efficient as deducing from a given string the next string in the enumeration.

```
NextString[s_String] := UnrankString[RankString[s] + 1]
```

```
NextString["AAAA"]
```

```
AAB
```

```
NestList[NextString, "", 15]
```

```
{, A, AA, B, AAA, AB, BA, C, AAAA, AAB, ABA, AC, BAA, BB, CA, D}
```

In practice it would be faster to iterate on the integer index and convert to the string for use. To use a different alphabet for this enumeration and those that follow, one need only change the definition of `characterWeights`.

To leave the alphabet unspecified (strings represented as lists of integers), one could remove the line in `RankString` and `StringN` (used by `UnrankString`) that does the replacements, but since one is then enumerating integer compositions, more appropriate names are indicated.

```
UnrankIntegerComposition[0] = {};
UnrankIntegerComposition[i_Integer] :=
 Module[{weight = BitLength[i], t = 1},
  Reap[Scan[If[# == 0, Sow[t]; t = 1, t++] &,
     IntegerDigits[i - 2^(weight - 1), 2, weight - 1]]; Sow[t]][[
   2, 1]]]
```

```
RankIntegerComposition[{}] = 0;
RankIntegerComposition[weight_List] :=
 FromDigits[
  Flatten[{1, Riffle[Table[1, {#}] & /@ (weight - 1), 0]}], 2]
```

■ A New String List Ordering

The string enumeration method described above can be modified to enumerate all *lists* of strings: Rather than using binary numbers of length $w-1$ to specify whether commas or pluses are placed between 1s, we can use ternary numbers and let the digits 0, 1, and 2 designate end-of-string, comma (end-of-character), or plus sign, respectively. This generates all possible lists of strings of total weight w. Again we must count how many codes are needed for the cases of weight up to w and add this to the index within the weight w

group. Any application that requires all lists of strings of all possible lengths containing all possible characters can use this technique.

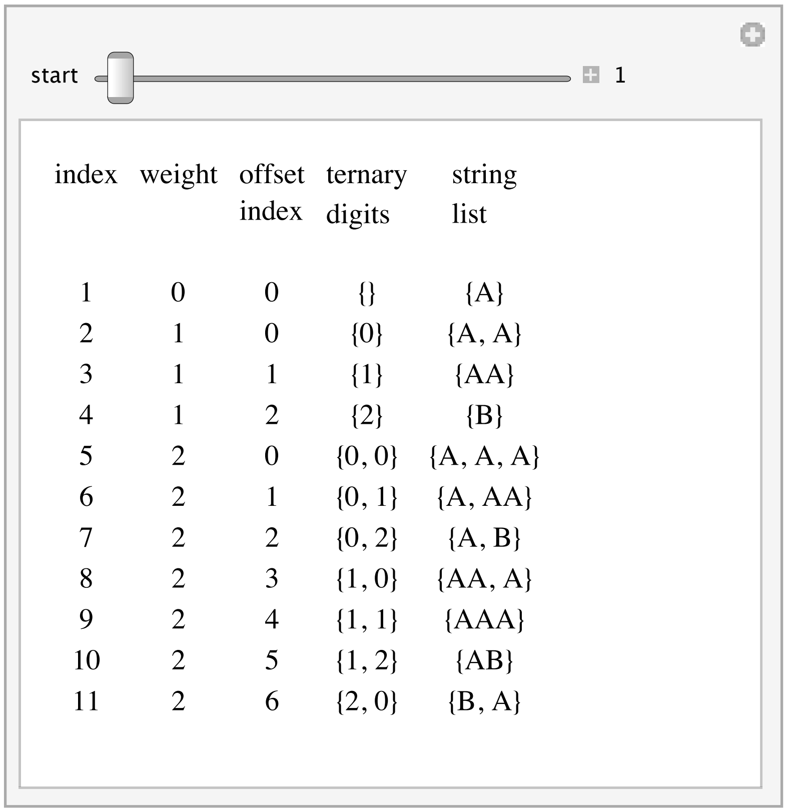
```
Sum[3^(k-1), {k, 1, n}] // TraditionalForm
```

$$\frac{1}{2}(3^n - 1)$$

```
Solve[i == %, n] // FullSimplify // TraditionalForm //
 Quiet
```

$$\left\{\left\{n \to \frac{\log(2\,i + 1)}{\log(3)}\right\}\right\}$$

Here is a `Manipulate` window showing the steps in converting an index into a string list.



start 1

index	weight	offset index	ternary digits	string list
1	0	0	{}	{A}
2	1	0	{0}	{A, A}
3	1	1	{1}	{AA}
4	1	2	{2}	{B}
5	2	0	{0, 0}	{A, A, A}
6	2	1	{0, 1}	{A, AA}
7	2	2	{0, 2}	{A, B}
8	2	3	{1, 0}	{AA, A}
9	2	4	{1, 1}	{AAA}
10	2	5	{1, 2}	{AB}
11	2	6	{2, 0}	{B, A}

We wrap this functionality in the function `UnrankStringList`. It first identifies the weight group n indicated by the index i (we will offset to make 1 rather than 0 the first index) and then subtracts $(3^n - 1)/2$ from it: this is the index within the weight group and is converted into a list of ternary digits. Starting with `{{1}}` (to represent a list containing a single string containing only character number 1, `A`), we scan through the list, taking appropriate action.

- 0 = end-of-string: end the string and start a new one, by appending `{1}` to `ans` (a new string `A`)
- 1 = end-of-character: end the character and start a new one, by appending `1` to the last part of `ans` (a new character `A`)
- 2 = increment character: increment the last character of the last string, by adding `1` to the last part of the last part of `ans`

```
UnrankStringList[i_Integer] :=
  Module[{n, j, maxDigit, ans = {{1}}},
   n = IntegerLength[2 i - 1, 3] - 1;
   (* Floor[Log[3,2 i-1]] *)
   j = (i - 1) - (3^n - 1) / 2;
   Scan[
    Switch[#,
      0, AppendTo[ans, {1}],
      1, AppendTo[ans[[-1]], 1],
      2, ans[[-1]][[-1]] ++
     ] &,
    IntegerDigits[j, 3, n]];
   maxDigit = Max[Flatten@ans];
   StringJoin @@@
    (ans /. (Reverse /@ characterWeights[[ ;; maxDigit + 1]]))];
```

We reverse the process to recover the index from the string list: first break strings into lists of characters and replace each character by its weight, then construct a ternary code from this list of lists of integers, then add the code found to the number of string lists of smaller weight with an offset to start the index at 1.

```
RankStringList[sl_List] :=
 Module[{wl, w, code = "", extrabit = 1},
  wl = (Characters /@ sl) //. characterWeights;
  w = Total[Flatten[wl]];
  While[wl ≠ {{1}},
   If[wl⟦-1⟧⟦-1⟧ > 1, code = "2" <> code; wl⟦-1⟧⟦-1⟧--,
    If[wl⟦-1⟧⟦-1⟧ == 1 && Length[wl⟦-1⟧] > 1,
     code = "1" <> code; wl⟦-1⟧ = Most[wl⟦-1⟧],
     If[wl⟦-1⟧ == {1}, code = "0" <> code;
      wl = Drop[wl, -1]]]]];
  FromDigits[code, 3] + (3^(w-1) - 1) / 2 + 1]
```

Here is how they work.

```
UnrankStringList /@ Range[13]

{{A}, {A, A}, {AA}, {B}, {A, A, A}, {A, AA},
 {A, B}, {AA, A}, {AAA}, {AB}, {B, A}, {BA}, {C}}
```

```
RankStringList /@ %

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}
```

In this implementation the functions only recognize the uppercase characters A–Z, but assuming access to some universal character list (not necessarily finite, just countable), all lists of all strings of all characters will appear in our list. Here is an example.

```
RankStringList@
 StringSplit@"ALL LISTS OF ALL STRINGS OF ALL CHARACTERS"

3 048 489 333 934 281 697 583 155 222 694 846 219 650 773 952 920 367 \
 806 711 282 418 052 558 080 535 064 728 674 326 377 083 360 908 188 \
 239 883 661 093 547 831 090 942 297 357 118 740 028 820 413 634 860 \
 300 163 055 898 102 115 650 978 443 282 394 869
```

```
UnrankStringList[%]

{ALL, LISTS, OF, ALL, STRINGS, OF, ALL, CHARACTERS}
```

Empty strings will not appear in the lists generated by `UnrankStringList`, and should not be included in input for `RankStringList`. To leave the alphabet unspecified and identify characters by number (weight) only, it is sufficient to remove the lines (first and last, respectively) in the definitions of `RankStringList` and `UnrankStringList` that do the replacements based on `characterWeights`.

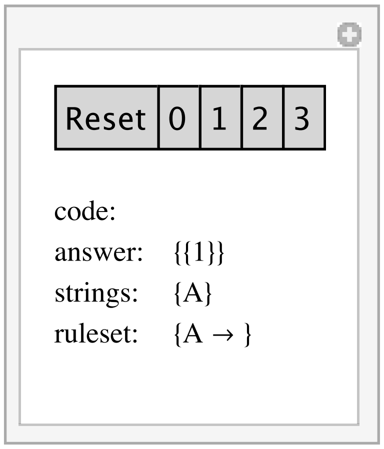
■ A New Ruleset Ordering

Sequential substitution system rulesets can be represented as a list of strings $\{s_1, s_2, s_3, s_4, \ldots, s_{2n}\}$, with an inferred meaning of $\{s_1 \to s_2, s_3 \to s_4, \ldots, s_{2n-1} \to s_{2n}\}$, a finite set of string replacement rules. In an SSS, a state string is first scanned for s_1, which if found is replaced by s_2. If no substring matching s_1 is found, the second replacement rule is invoked, and if needed, the third rule, etc. At each step, only the first possible replacement of the first matching rule is performed. (For more details concerning sequential substitution systems, see Chapter 3, Section 6 of *A New Kind of Science* [1].)

If the string list contains an odd number of strings, we have the option of simply throwing away the last string, but that would result in many duplicate rulesets, and there may be a way to use the extra information in the last half rule. One way would be to append a final empty string and thus create in these cases a final rule *string* → `""`. In fact, such "something to nothing" rules, rules that delete a specified substring, are never generated otherwise by the ternary index algorithm, but *should* be allowed at any position in the ruleset. `UnrankString` only generated the empty string because of a separate rule defined for the input 0. In the same way, some separate rule or algorithm could be used to insert empty strings in the string list to allow *string* → `""` and `""` → *string* rules, creating additional rulesets with the same total weight. So our simple list of string lists of a given weight should be extended by the insertion of empty strings—but should they be allowed at all positions? Let us consider the ramifications of including empty strings for a moment.

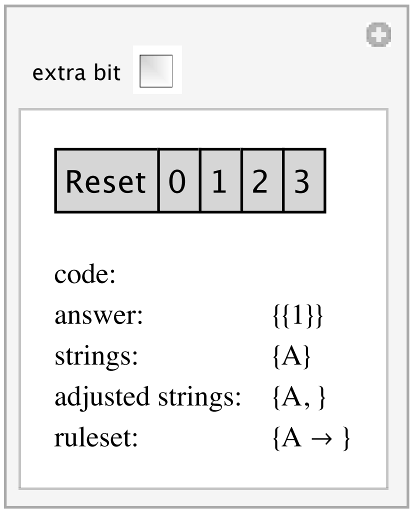
- As mentioned, "something to nothing" rules simply delete a specified substring. This occurs when an empty string is in an even position in the string list. For completeness we need all possible cases of this type.
- On the other hand "nothing to something" rules, rules of the form `""` → *string*, will always match at the very beginning of the state string: Look for nothing (we will always find it) and insert a given string there—this is basically an insertion rule. (Note that the insertion always occurs at the beginning of the string, but rules such as `"A"` → `"AB"` effectively cause later insertions.) Now if included, a "nothing to something" rule, an initial insertion rule, should always be the last rule of the ruleset since any following rules would never be invoked. Therefore we only need an empty string at an odd position in the string list when that is the next-to-last position.

It is probably more trouble than it is worth to optionally insert empty strings at only the positions that are "even or next-to-last" in the string list, but these two criteria suggest a simple way to make sure that *at least* these cases are included while ruling out many of the unwanted cases. Since empty strings never need occur at the beginning of the string list (unless that is in fact the next-to-last position), we will consider inserting an empty string as an alternative way of ending the previous string and starting another (with an empty string inserted between). This will also guarantee that empty strings are not inserted more frequently than every other position, further reducing their occurrence at undesired positions. In order to allow a “something to nothing” rule at the end of the ruleset, if the number of rules is odd we will interpret the final string as such a rule. This method will give some cases with empty strings at odd positions earlier in the list, but we will drop them, and they will occur far less often than if all possible empty string insertions were allowed. It is also trivial to add the new instruction by simply switching to quaternary representation of the index and letting the digits 0, 1, 2, and 3 designate end-string-insert-empty-string-and-start-next-string, end-string-and-start-next-string, comma (end-character-and-start-next-character), or plus sign, respectively. The new instruction forces an immediate end of the previous string, just as does the end-of-string symbol. This generates almost all possible useful sequences of strings of (total sequence) weight *w*, now including empty strings at all possible positions—except at the beginning of the ruleset. Unfortunately we will get empty strings at some undesired positions, such as in nonfinal "" → *string* rules, but disallowing them at consecutive positions is good. In any case, the whole point is that we at least get all the cases we want to include automatically. Is that true here?



code:
answer: {{1}}
strings: {A}
ruleset: {A → }

Unfortunately not. A little experimentation shows that we easily get “something to nothing” rules both singly and as part of a larger ruleset, but “nothing to something” rules never appear alone. We need some additional way to optionally insert an empty string in the next-to-last position. It is enough to add one extra bit that can be appended to the code.



```
code:
answer:             {{1}}
strings:            {A}
adjusted strings:   {A, }
ruleset:            {A → }
```

This does create more duplicates (such as code 1 with extra bit, code 0 without: both give `{"A" → "", "A" → ""}`), but all the ones we want are there. To create rank and unrank functions we will need to know how many rulesets there are of weight less than k.

```
1 + Sum[4^(k-1), {k, 1, n}] // Simplify // TraditionalForm
```

$$\frac{1}{3}(4^n + 2)$$

```
Solve[i == %, n] // FullSimplify // TraditionalForm //
 Quiet
```

$$\left\{\left\{n \to \frac{\log(3\, i - 2)}{\log(4)}\right\}\right\}$$

Here is the unrank function.

```
UnrankRuleset[iplusflag_Integer /; iplusflag > 0] :=
  Module[{i, extraflag, n, j, quaternaryDigits, maxDigit,
    ans = {{1}}, strings},
   extraflag = OddQ[iplusflag];
   i = Quotient[iplusflag + 1, 2];
   n = IntegerLength[3 i - 2, 4] - 1;
   (* Floor[Log[4,3 i-2]] *)
   j = i - (4^n + 2) / 3;
   quaternaryDigits = IntegerDigits[j, 4, n];
   Scan[Switch[#, 0 , ans = Join[ans, {{}, {1}}] , 1,
      AppendTo[ans, {1}], 2, AppendTo[ans[[-1]], 1], 3,
      ans[[-1]][[-1]]++] &, quaternaryDigits];
   maxDigit = Max[Flatten@ans];
   strings = StringJoin @@@
     (ans /.
       (Reverse /@ characterWeights[[ ;; maxDigit + 1]]));
   If[extraflag,
    strings = Join[Most[strings], {"", Last[strings]}]];
   If[OddQ[Length[strings]],
    strings = AppendTo[strings, ""]];
   Rule @@@ Partition[strings, 2, 2]];
```

Notice that rulesets are grouped by total weight. First come the rulesets of weight 1, 2, and 3.

```
UnrankRuleset /@ Range[2]
```

```
{{ → A}, {A → }}
```

```
UnrankRuleset /@ Range[3, 10]
```

```
{{A → ,  → A}, {A → , A → }, {A → , A → },
 {A → A}, { → AA}, {AA → }, { → B}, {B → }}
```

```
UnrankRuleset /@ Range[11, 42]
```

```
{{A → , A → ,  → A}, {A → , A → , A → }, {A → , A → , A → },
 {A → , A → A}, {A → ,  → AA}, {A → , AA → }, {A → ,  → B},
 {A → , B → }, {A → A,  → , A → }, {A → A,  → A}, {A → A,  → A},
 {A → A, A → }, {A → , AA → }, {A → AA}, {A → , B → }, {A → B},
 {AA → ,  → A}, {AA → , A → }, {AA → , A → }, {AA → A}, { → AAA},
 {AAA → }, { → AB}, {AB → }, {B → ,  → A}, {B → , A → },
 {B → , A → }, {B → A}, { → BA}, {BA → }, { → C}, {C → }}
```

Within each weight grouping the rulesets are sorted preferentially to have short strings and secondarily to have low-weight characters early in the string list.

There are some duplicates, but we will be able to discard them when we have an inverse function; if the index is different from the one we started with, it is a duplicate. In any case there will be others to discard as well, rulesets where certain rules will never be invoked—functional duplicates of previous rulesets—that can be discarded out of hand without using them. These include:

1. Any case including an identity rule, such as A → A. If this rule is ever invoked, it will continue to be invoked thereafter and no further changes will occur.
2. Any case including a nonfinal rule "" → *string*. This rule prevents subsequent rules from ever being invoked.
3. Any case with two rules with the same left-hand side, such as {A → B, A → C}. The second rule will never be invoked and the ruleset will thus be a duplicate of a simpler ruleset.
4. Any case with two rules with left-hand sides s_1 and s_2, such that s_1 is a substring of s_2. The second rule will never be invoked and so this ruleset is an effective duplicate.

In fact, it can be seen that 2 and 3 are special cases of 4. Of course, it would be preferable to generate rulesets that do not include such cases, but it is not immediately obvious how to do so. In any case, much of the redundancy *has* been eliminated, and what remains is understood and the redundant rulesets identified and skipped over. We have a clear way of iterating through rulesets, although not all of them will be used, and of course, even among those used there will be many duplicate sequential substitution system graphs (from permuting the order of the characters, for instance).

```
RankRuleset[rs_List] :=
 Module[{rl, wl, w, code = "", extrabit = 1},
  rl = Flatten[List @@@ rs];
  If[Last[rl] == "", rl = Most[rl]];
  If[Length[rl] > 1 && rl[[-2]] == "", extrabit = 0;
   rl = Drop[rl, {-2}]];
  wl = (Characters /@ rl) //. characterWeights;
  w = Total[Flatten[wl]];
  While[wl ≠ {{1}},
   If[wl[[-1]][[-1]] > 1, code = "3" <> code; wl[[-1]][[-1]]--,
    If[wl[[-1]][[-1]] == 1 && Length[wl[[-1]]] > 1,
     code = "2" <> code; wl[[-1]] = Most[wl[[-1]]],
     If[Length[wl] ≥ 2 && wl[[-2 ;;]] == {{}, {1}},
      code = "0" <> code; wl = Drop[wl, -2],
      If[wl[[-1]] == {1}, code = "1" <> code;
       wl = Drop[wl, -1]]]]]];
  2 (FromDigits[code, 4] + (4^(w-1) + 2) / 3) + extrabit - 1]
```

The final line adds the number of rulesets of smaller weights to the reconstructed quaternary code, left-shifts the result, adds the extra bit flag at the end, and finally subtracts 1 in order to start at 1 instead of at 2. Again, other alphabets may be accommodated by adjusting `characterWeights`, an unspecified alphabet by removing the replacement code.

The following tests the ruleset rank and unrank functions:

```
RankRuleset[{"ABA" → "A", "CAA" → "ABC", "" → "AB",
  "AAB" → "A"}]
```

```
148 889 211 045 382
```

```
UnrankRuleset[%]
```

```
{ABA → A, CAA → ABC,  → AB, AAB → A}
```

```
UnrankRuleset /@ Range[25]
```

```
{{ → A}, {A → }, {A → ,  → A}, {A → , A → },
 {A → , A → }, {A → A}, { → AA}, {AA → }, { → B}, {B → },
 {A → , A → ,  → A}, {A → , A → , A → }, {A → , A → , A → },
 {A → , A → A}, {A → ,  → AA}, {A → , AA → }, {A → ,  → B},
 {A → , B → }, {A → A,  → , A → }, {A → A,  → A}, {A → A,  → A},
 {A → A, A → }, {A → , AA → }, {A → AA}, {A → , B → }}
```

```
RankRuleset /@ %
```

```
{1, 2, 3, 5, 5, 6, 7, 8, 9, 10, 11, 13, 13,
 14, 15, 23, 17, 25, 19, 21, 21, 22, 23, 24, 25}
```

Now we create an increment function to advance to the next (unique and useful) ruleset. Initially we only test whether `RankRuleset` returns the same index, but we already include the option of additional tests.

```
NextRuleset[n_Integer] := Module[{k = 1, nrs},
   While[nrs = UnrankRuleset[n + k];
    RankRuleset[nrs] ≠ n + k || ! UsefulRulesetQ[nrs], k++];
   nrs];
NextRuleset[rs_List] := NextRuleset[RankRuleset[rs]];
```

```
UsefulRulesetQ[rs_List] = True;
```

```
NestList[NextRuleset, UnrankRuleset@1, 50]
```

```
{{ → A}, {A → }, {A → ,  → A}, {A → , A → }, {A → A}, { → AA},
 {AA → }, { → B}, {B → }, {A → , A → ,  → A}, {A → , A → , A → },
 {A → , A → A}, {A → ,  → AA}, {A → ,  → B}, {A → A,  → , A → },
 {A → A,  → A}, {A → A, A → }, {A → , AA → }, {A → AA}, {A → , B → },
 {A → B}, {AA → ,  → A}, {AA → , A → }, {AA → A}, { → AAA},
 {AAA → }, { → AB}, {AB → }, {B → ,  → A}, {B → , A → }, {B → A},
 { → BA}, {BA → }, { → C}, {C → }, {A → , A → , A → ,  → A},
 {A → , A → , A → , A → }, {A → , A → , A → A}, {A → , A → ,  → AA},
 {A → , A → ,  → B}, {A → , A → A,  → , A → }, {A → , A → A,  → A},
 {A → , A → A, A → }, {A → , A → , AA → }, {A → , A → AA},
 {A → , A → , B → }, {A → , A → B}, {A → , AA → ,  → A},
 {A → , AA → , A → }, {A → , AA → A}, {A → ,  → AAA}}
```

Here are their indices.

```
RankRuleset /@ %
```

```
{1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15,
 17, 19, 21, 22, 23, 24, 25, 26, 27, 29, 30, 31,
 32, 33, 34, 35, 37, 38, 39, 40, 41, 42, 43, 45, 46,
 47, 49, 51, 53, 54, 55, 56, 57, 58, 59, 61, 62, 63}
```

Here are the duplicates.

```
Complement[Range[Last@%], %]
```

```
{4, 12, 16, 18, 20, 28, 36, 44, 48, 50, 52, 60}
```

```
UnrankRuleset /@ %
```

```
{{A → , A → }, {A → , A → , A → }, {A → , AA → },
 {A → , B → }, {A → A,  → A}, {AA → , A → }, {B → , A → },
 {A → , A → , A → , A → }, {A → , A → , AA → },
 {A → , A → , B → }, {A → , A → A,  → A}, {A → , AA → , A → }}
```

Yes, these are mostly rulesets with final “something to nothing” rules. Well, it is the price of doing business; the important thing is to include all those needed.

Now we restrict the returned values to rulesets that generate different SSSs by discarding cases with: (1) an identity rule; or (2) two rules with left-hand sides s_1 and s_2, such that s_1 is a substring of s_2.

```
Clear[UsefulRulesetQ];
UsefulRulesetQ[rs_List] :=
 Module[{lhs, max, i, j, dup},
  If[(Or @@ (Equal @@@ rs)), dup = True,
   If[(Length[rs] == 1), dup = False,
    lhs = First /@ rs;
    max = Length[lhs];
    i = 1; j = 2; dup = False;
    While[! dup && i < max,
     If[Length[StringPosition[lhs⟦j⟧, lhs⟦i⟧]] > 0,
      dup = True];
     j ++;
     If[j > max, i ++; j = i + 1]]]];
  ! dup]
```

Now we can use `NextRuleset` to iterate through all valid rulesets giving useful SSSs. All problem cases have been eliminated, and all other rulesets will appear somewhere in the list. Here are the first 50.

```
NestList[NextRuleset, UnrankRuleset@1, 49]
```

```
{{ → A}, {A → }, {A → ,  → A}, { → AA}, {AA → }, { → B}, {B → },
 {A → ,  → AA}, {A → ,  → B}, {A → AA}, {A → , B → }, {A → B},
 {AA → ,  → A}, {AA → , A → }, {AA → A}, { → AAA}, {AAA → },
 { → AB}, {AB → }, {B → ,  → A}, {B → , A → }, {B → A}, { → BA},
 {BA → }, { → C}, {C → }, {A → ,  → AAA}, {A → ,  → AB},
 {A → , B → ,  → A}, {A → , B → A}, {A → ,  → BA}, {A → ,  → C},
 {A → AA,  → A}, {A → AAA}, {A → AB}, {A → B,  → A}, {A → BA},
 {A → , C → }, {A → C}, {AA → , A → ,  → A}, {AA → ,  → AA},
 {AA → ,  → B}, {AA → A,  → A}, {AA → A, A → }, {AA → , B → },
 {AA → B}, {AAA → ,  → A}, {AAA → , A → }, {AAA → A}, { → AAAA}}
```

Here are their indices.

```
RankRuleset /@ %
```

```
{1, 2, 3, 7, 8, 9, 10, 15, 17, 24, 25, 26, 27, 29, 30,
 31, 32, 33, 34, 35, 37, 38, 39, 40, 41, 42, 63, 65,
 67, 70, 71, 73, 93, 96, 98, 101, 104, 105, 106, 107,
 111, 113, 117, 118, 121, 122, 123, 125, 126, 127}
```

Here are the duplicates and functional duplicates that were skipped.

```
Complement[Range[Last@%], %]
```

```
{4, 5, 6, 11, 12, 13, 14, 16, 18, 19, 20, 21, 22, 23,
 28, 36, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53,
 54, 55, 56, 57, 58, 59, 60, 61, 62, 64, 66, 68, 69,
 72, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85,
 86, 87, 88, 89, 90, 91, 92, 94, 95, 97, 99, 100, 102,
 103, 108, 109, 110, 112, 114, 115, 116, 119, 120, 124}
```

```
UnrankRuleset /@ %
```

```
{{A→ , A→ }, {A→ , A→ }, {A→A}, {A→ , A→ , →A},
 {A→ , A→ , A→ }, {A→ , A→ , A→ }, {A→ , A→A},
 {A→ , AA→ }, {A→ , B→ }, {A→A, → , A→ }, {A→A, →A},
 {A→A, →A}, {A→A, A→ }, {A→ , AA→ }, {AA→ , A→ },
 {B→ , A→ }, {A→ , A→ , A→ , →A}, {A→ , A→ , A→ , A→ },
 {A→ , A→ , A→ , A→ }, {A→ , A→ , A→A}, {A→ , A→ , →AA},
 {A→ , A→ , AA→ }, {A→ , A→ , →B}, {A→ , A→ , B→ },
 {A→ , A→A, → , A→ }, {A→ , A→A, →A}, {A→ , A→A, →A},
 {A→ , A→A, A→ }, {A→ , A→ , AA→ }, {A→ , A→AA},
 {A→ , A→ , B→ }, {A→ , A→B}, {A→ , AA→ , →A},
 {A→ , AA→ , A→ }, {A→ , AA→ , A→ }, {A→ , AA→A},
 {A→ , AAA→ }, {A→ , AB→ }, {A→ , B→ , A→ },
 {A→ , B→ , A→ }, {A→ , BA→ }, {A→ , C→ },
 {A→A, →A, → , A→ }, {A→A, →A, →A}, {A→A, →A, →A},
 {A→A, →A, A→ }, {A→A, → , AA→ }, {A→A, →AA},
 {A→A, → , B→ }, {A→A, →B}, {A→A, A→ , →A},
 {A→A, A→ , A→ }, {A→A, A→ , A→ }, {A→A, A→A},
 {A→A, →AA}, {A→A, AA→ }, {A→A, →B}, {A→A, B→ },
 {A→AA, → , A→ }, {A→AA, →A}, {A→AA, A→ }, {A→ , AAA→ },
 {A→ , AB→ }, {A→B, → , A→ }, {A→B, →A}, {A→B, A→ },
 {A→ , BA→ }, {AA→ , A→ , A→ }, {AA→ , A→ , A→ },
 {AA→ , A→A}, {AA→ , AA→ }, {AA→ , B→ }, {AA→A, → , A→ },
 {AA→A, →A}, {AA→ , AA→ }, {AA→AA}, {AAA→ , A→ }}
```

```
RankRuleset /@ %
```

```
{5, 5, 6, 11, 13, 13, 14, 23, 25, 19, 21, 21, 22, 23,
 29, 37, 43, 45, 45, 46, 47, 55, 49, 57, 51, 53, 53,
 54, 55, 56, 57, 58, 59, 61, 61, 62, 95, 97, 69, 69,
 103, 105, 75, 77, 77, 78, 79, 87, 81, 89, 83, 85, 85,
 86, 87, 88, 89, 90, 91, 93, 94, 95, 97, 99, 101, 102,
 103, 109, 109, 110, 119, 121, 115, 117, 119, 120, 125}
```

Some are true duplicates (if the index number changed), the others functional duplicates containing rules that will never be invoked in creating a sequential substitution system. This is a manageable situation. Remember, Cantor's diagonalization contains an infinite number of duplicates of each fraction! The following table summarizes the numbers of unique rulesets and useful (functionally distinct) rulesets for several orders of magnitude. Roughly three-fourths of the generated rulesets are unique, although fewer and fewer of the unique rulesets will give distinct sequential substitution systems.

```
Text[
 Table[With[{n = 10^k},
    With[{l = DeleteDuplicates[UnrankRuleset /@ Range[n]]},
     Flatten[{#, Round[100.0 Rest[#] / n, 1]}] & @
      {n, Length@l, Length[Select[l, UsefulRulesetQ]]}
    ]], {k, 1, 6}] //
  TableForm[#, TableHeadings →
     {None, {Style["n", Italic], "unique", "useful",
       "% unique", "% useful"}}, TableAlignments → Right] &]
```

n	unique	useful	% unique	% useful
10	9	7	90	70
100	80	37	80	37
1000	766	255	77	26
10 000	7550	1970	76	20
100 000	75 177	8267	75	8
1 000 000	750 588	66 775	75	7

Conclusion

A review of three methods for enumerating the rational numbers motivated the development of a bijective enumeration of arbitrary-length strings on a countable alphabet of characters (or equivalently, an enumeration of integer compositions). This in turn was extended to form an enumeration of all finite-length sequences of finite-length strings, and an enumeration of all sequential substitution system rulesets. The latter list, although including both exact and functional duplicates, is well defined and relatively dense. Rank, unrank, and successor functions were discussed.

Acknowledgments

My thanks to the NKS Summer School 2009 staff and in particular to Matthew Szudzik for encouraging my interest in enumerations and sequential substitution systems, and to Charles Sarr for his willingness to discuss these ideas at considerable length.

■ References

[1] S. Wolfram, *A New Kind of Science*, Champaign, IL: Wolfram Media, 2002.

[2] T. Rowland. "Enumerating Strings" from *The NKS Forum*—A Wolfram Web Resource. (Apr 02, 2010) forum.wolframscience.com/showthread.php?s=&threadid=929.

[3] G. Beck. "A Path through the Lattice Points in a Quadrant" from the Wolfram Demonstrations Project—A Wolfram Web Resource. www.demonstrations.wolfram.com/APathThroughTheLatticePointsInAQuadrant.

[4] D. Bradley. "Counting the Positive Rationals: A Brief Survey." (Jun 21, 2010) arxiv.org/abs/math/0509025.

[5] Y. Sagher, "Counting the Rationals," *American Mathematical Monthly*, **96**(9), 1989 p. 823.

[6] N. Calkin and H. Wilf, "Recounting the Rationals," *American Mathematical Monthly*, **107**(4), 2000 pp. 360–363. www.math.upenn.edu/~wilf/reprints.html.

[7] D. Knuth, C. Rupert, A. Smith, and R. Stong, "Recounting the Rationals, Continued," *American Mathematical Monthly*, **110**(7), 2003 pp. 642–643.

[8] J. Czyz and W. Self, "The Rationals Are Countable: Euclid's Proof," *College Mathematics Journal*, **34**(5), 2003 pp. 367–369.

[9] M. Szudzik. "Enumerating the Rationals" from the Wolfram Demonstrations Project—A Wolfram Web Resource. www.demonstrations.wolfram.com/EnumeratingTheRationals.

[10] B. Yorgey, "Recounting the Rationals, Part II," *The Math Less Traveled* (blog), (Apr 2, 2010) www.mathlesstraveled.com/?p=97.

[11] E. Weisstein. "Composition" from Wolfram *MathWorld*—A Wolfram Web Resource. www.mathworld.wolfram.com/Composition.html.

[12] S. Heubach and T. Mansour, *Combinatorics of Compositions and Words*, Boca Raton, FL: CRC Press, 2009.

[13] K. Caviness. "Universal String Enumeration" from the Wolfram Demonstrations Project—A Wolfram Web Resource. www.demonstrations.wolfram.com/UniversalStringEnumeration.

[14] R. Kantrowitz, "A Principle of Countability," *Mathematics Magazine*, **73**(1), 2000 pp. 40–42.

K. E. Caviness, "Indexing Strings and Rulesets," *The Mathematica Journal*, 2011.
dx.doi.org/doi:10.3888/tmj.13–6.

About the Author

Ken Caviness teaches physics at Southern Adventist University, a small liberal arts university near Chattanooga, Tennessee. He holds a Ph.D. in physics (emphases in relativity and nuclear physics) from the University of Massachusetts at Lowell, and has taught math and physics in Rwanda, Texas, and Tennessee. His interests include both computer and human languages (including the planned language Esperanto). He has used *Mathematica* since Version 1, both professionally and for recreational programming.

Kenneth E. Caviness
Physics Department
Southern Adventist University
PO Box 370, Collegedale, TN 37315-0370
caviness@southern.edu