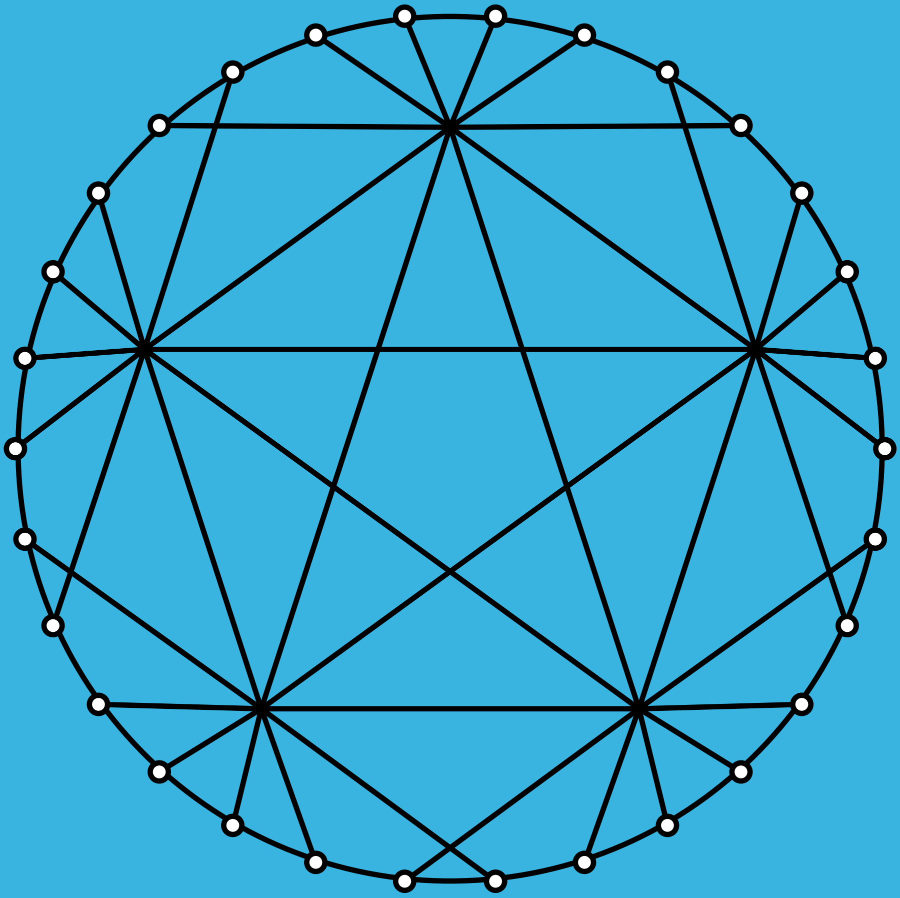
BULLETIN of the INSTITUTE of COMBINATORICS and its APPLICATIONS

Volume 86
June 2019

Editors-in-Chief: Marco Buratti, Donald Kreher, Tran van Trung



Boca Raton, FL, U.S.A.

ISSN 1182 - 1278

BULLETIN OF THE ICA
Volume **86** (2019), Pages 101–122

Successor algorithms via counting arguments

Nicholas A. Loehr

Virginia Tech, Blacksburg, Virgina, U.S.A.
nloehr@vt.edu

Abstract: A successor algorithm for a set of combinatorial objects takes as input an arbitrary object in the set and computes the next object relative to some total ordering. This note presents an extremely simple general paradigm for creating successor algorithms based on three fundamental counting rules (the Bijection Rule, the Sum Rule, and the Product Rule). Using this approach, we can proceed immediately and automatically from a counting argument for a given set into a successor algorithm for that set. We show that this technique applies to many classical combinatorial families such as words, subsets, permutations, anagrams, set partitions, integer partitions, trees, tableaux, and derangements. We also compare our method to two other general approaches to successor algorithms: decision trees and the Nijenhuis-Wilf combinatorial family construction. Finally, we present versions of the three counting rules that convert counting arguments to ranking and unranking algorithms for finite sets.

Key words and phrases: successor algorithms, combinatorial enumeration, counting.

AMS (MOS) Subject Classifications: 05-04, 68R05, 05A15

Received: 3 January 2019
Accepted: 23 April 2019

1 Introduction

An important problem in the theory of combinatorial algorithms is to create programs that generate a list of all combinatorial objects of a given kind. For example, we may seek algorithms to generate words, subsets, permutations, anagrams, integer partitions, set partitions, trees, tableaux, etc., satisfying given restrictions. Subroutines for listing such objects appear in many standard references including the books by Nijenhuis and Wilf [5, 10], Knuth [3, Sec. 7.2], Stanton and White [9], Bender and Williamson [1], Reingold, Nievergelt, and Deo [7], Kreher and Stinson [4], and Ruskey [8].

We can think of an algorithm for listing the objects in a given set S as consisting of two subroutines (depending on the set S) called $\textsc{first}$ and $\textsc{next}$. Calling the routine $\textsc{first}(S)$ returns the first object in S relative to some (explicit or implicit) total ordering of S. Given a particular object $x \in S$, calling the routine $\textsc{next}(x, S)$ returns the immediate successor of x in S relative to the ordering. If x happens to be the last object in S, the routine $\textsc{next}$ returns a special value `LAST`. We can now list all objects in S using a loop that initializes x to be $\textsc{first}(S)$, then repeatedly replaces x by $\textsc{next}(x, S)$ until the `LAST` flag is returned.

The purpose of this note is to describe a simple general framework for automatically converting *counting arguments* into *successor algorithms.* We begin by recalling three fundamental counting rules that form the very foundation for enumerative combinatorics.

Bijection Rule. If $F : S \to T$ is a bijection (i.e., a function that is one-to-one and onto), then

$$|S| = |T|.$$

Sum Rule. If $S_1, S_2, \ldots, S_k$ are pairwise disjoint finite sets, then

$$|S_1 \cup S_2 \cup \cdots \cup S_k| = |S_1| + |S_2| + \cdots + |S_k|.$$

Product Rule. If $S_1, S_2, \ldots, S_k$ are finite sets, then

$$|S_1 \times S_2 \times \cdots \times S_k| = |S_1| \cdot |S_2| \cdot \ldots \cdot |S_k|.$$

In Section 2, we convert these basic counting rules into *Successor Rules* for building successor algorithms. The main point is that anytime one can *count* a set S using the three counting rules, one can automatically create a *successor algorithm* for S with no further effort. In particular, one need not give separate, ad hoc treatments for each different family of combinatorial objects (permutations, subsets, partitions, etc.), as is often done in the literature on this subject. All algorithms can be developed uniformly within the single unifying framework provided by the successor rules.

In Section 3, we give examples of successor algorithms created from counting arguments via the successor rules. In particular, we construct algorithms for listing subsets, anagrams, set partitions, and derangements. We also briefly discuss other families of objects to which this technique applies, including permutations, integer partitions, trees, and tableaux. We stress that the main contribution is not the individual successor algorithms presented as examples, but rather the extremely simple *meta-algorithm* by which these algorithms are automatically created. In Section 4, we compare our method to two other general approaches to this problem, decision trees and the "combinatorial families" defined in [5, Chpt. 13].

In Section 5, we briefly discuss the related problem of creating ranking and unranking algorithms for finite sets. We develop versions of the three fundamental counting rules that automatically convert counting arguments to ranking and unranking algorithms for the sets being counted.

2 The successor rules

The Bijection Rule, Sum Rule, and Product Rule allow us to count finite sets. We now present the corresponding Successor Rules for building successor algorithms for these sets. The rules operate recursively, assembling FIRST and NEXT subroutines for previously studied sets to create FIRST and NEXT subroutines for new sets.

Successor Bijection Rule. Let $F : S \to T$ be a bijection with inverse $G : T \to S$, and suppose we already know successor subroutines for the set S. Then T has successor subroutines defined as follows: $\textsc{first}(T) = F(\textsc{first}(S))$ and $\textsc{next}(y, T) = F(\textsc{next}(G(y), S))$. Here we use the convention $F(\textsc{last}) = \textsc{last}$.

Successor Sum Rule. Let $(S_1, S_2, \ldots, S_k)$ be a given sequence of nonempty pairwise disjoint finite sets for which successor subroutines FIRST and NEXT are already known. Then $S = S_1 \cup S_2 \cup \cdots \cup S_k$ has successor subroutines defined as follows:

```
procedure first(S)
{ return first(S_1); }

procedure next(x,S)
{ find the unique i with x in S_i;
  y=next(x,S_i);
  if (y is not LAST) return y;
  if (i<k) return first(S_{i+1});
  return LAST;
}
```

The pseudocode can readily be modified to account for the case where some sets S_i may be empty. For example, the FIRST subroutine for S would return $\textsc{first}(S_j)$, where j is the minimal index with $S_j \neq \emptyset$.

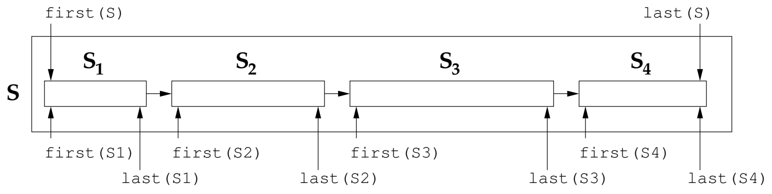


Figure 1: Schematic diagram for the Successor Sum Rule.

Figure 1 gives a visual representation of how the Successor Sum Rule operates. Intuitively, this rule lists elements of S by first listing objects in S_1, then listing objects in S_2, and so on, finally listing all objects in S_k. Each subset is listed recursively using successor subroutines already available for the sets S_j. Reordering the terms in the sequence $(S_1, S_2, \ldots, S_k)$ leads to other successor algorithms for S that list the objects of S in different orders.

Successor Product Rule. Let $S_1, S_2, \ldots, S_k$ be given nonempty finite sets for which successor subroutines FIRST and NEXT are already known. Then $S = S_1 \times S_2 \times \cdots \times S_k$ has successor subroutines defined as follows:

```
procedure first(S)
{ return (first(S_1),first(S_2),...,first(S_k)); }

procedure next(x,S)
{ write x=(x_1,x_2,...,x_k),
                   where x_i is in S_i for each i;
  j=k;
  while (j>0 and next(x_j,S_j) is LAST) do j=j-1;
  if (j is 0) return LAST;
  return (x_1,...,x_{j-1},next(x_j,S_j),
                   first(S_{j+1}),...,first(S_k));
}
```

Here is an intuitive description of how the Successor Product Rule operates. The NEXT routine for S acts like an odometer counting up through the values of the k-tuple $(x_1, x_2, \ldots, x_k)$, using the NEXT routines for each S_i to increment each "digit" x_i. In a given call to NEXT for S, the routine first finds the index j such that x_j is not last in S_j, but all values beyond x_j are last in their respective sets. To proceed, x_j is incremented and all later values are reset to the first positions in their respective sets. This succeeds unless all x_i are last in their sets, in which case we return LAST.

Remarkably, we can actually construct the Successor Product Rule automatically using the more primitive Successor Sum Rule. We briefly sketch how this is done. Given a product set $S \times T$ with only two factors, let $S = \{s_1, \ldots, s_m\}$. Write $S \times T$ as the disjoint union of m sets $T_1, \ldots, T_m$, where $T_i = \{s_i\} \times T$. A successor algorithm for T induces a successor algorithm for each T_i via the Successor Bijection Rule. Specifically, $\textsc{first}(T_i) = (s_i,\textsc{first}(T))$ and $\textsc{next}((s_i,t),T_i) = (s_i,\textsc{next}(t,T))$. The successor algorithms for $T_1,\ldots,T_m$ get linked together by the Successor Sum Rule to give a successor algorithm for $S \times T$. One readily checks that this algorithm acts exactly like the algorithm in the Successor Product Rule when $k = 2$. The algorithm for general k follows by induction on k, thinking of the Cartesian product $S_1 \times \cdots \times S_{k+1}$ of $k+1$ factors as the product of the two sets S_1 and $S_2 \times \cdots \times S_{k+1}$. Similarly, the Successor Sum Rule for k sets can be constructed automatically by induction from the Successor Sum Rule for two sets.

3 Examples

This section gives examples of how counting arguments can be automatically converted to successor algorithms by application of the Successor Rules. A key point is that one often needs to indicate explicitly how the Bijection Rule is used in the original counting argument, so that the associated successor subroutines perform the correct bookkeeping tasks on intermediate objects. Throughout this section, we write $[n]$ to denote the set $\{1,2,\ldots,n\}$.

3.1 Subsets

For $0 \leq k \leq n$, let $\mathrm{Sub}(n,k)$ be the set of all k-element subsets of $[n]$. It is well known that $|\mathrm{Sub}(n,k)|$ is the binomial coefficient $\binom{n}{k}$. Furthermore, the numbers $|\mathrm{Sub}(n,k)|$ are characterized by the "Pascal's Triangle" recursion

$$|\mathrm{Sub}(n,k)| = |\mathrm{Sub}(n-1,k)| + |\mathrm{Sub}(n-1,k-1)|, \qquad (1)$$

for $0 < k < n$, with initial conditions $|\mathrm{Sub}(n,0)| = |\mathrm{Sub}(n,n)| = 1$ for all $n \geq 0$.

The standard proof of (1) decomposes $\mathrm{Sub}(n,k)$ into the disjoint union of sets A and B, where $A = \mathrm{Sub}(n-1,k)$ is the set of k-element subsets of $[n]$ that do not contain n, and B is the set of k-element subsets of $[n]$ that do contain n. Deleting n from each subset in B defines a bijection from B onto $\mathrm{Sub}(n-1,k-1)$. Now (1) follows immediately from the Bijection Rule and the Sum Rule.

Applying the Successor Bijection Rule and the Successor Sum Rule, we automatically obtain recursively defined successor algorithms for all the sets $\mathrm{Sub}(n,k)$. For the base case $k=0$, $\textsc{first}(\mathrm{Sub}(n,0))$ returns the empty subset and $\textsc{next}(\emptyset, \mathrm{Sub}(n,0))$ returns $\textsc{last}$. For the base case $k=n$, $\textsc{first}(\mathrm{Sub}(n,n))$ returns the subset $[n]$ and $\textsc{next}([n], \mathrm{Sub}(n,n))$ returns $\textsc{last}$. When $0<k<n$, the routine $\textsc{first}(\mathrm{Sub}(n,k))$ recursively returns $\textsc{first}(\mathrm{Sub}(n-1,k))$. Given $T \in \mathrm{Sub}(n,k)$, we compute $\textsc{next}(T, \mathrm{Sub}(n,k))$ as follows.

Case 1: $n \notin T$. Return the answer $\textsc{next}(T, \mathrm{Sub}(n-1,k))$ unless this answer is $\textsc{last}$, in which case return $\textsc{first}(\mathrm{Sub}(n-1,k-1))$ with n adjoined.

Case 2: $n \in T$. Let T' be T with n deleted. If $\textsc{next}(T', \mathrm{Sub}(n-1,k-1))$ is not $\textsc{last}$, then return this answer with n adjoined; otherwise return $\textsc{last}$.

This successor algorithm lists all k-element subsets of $[n]$ not containing n first, followed by all k-element subsets of $[n]$ that do contain n. Interchanging the two terms on the right side of (1) would lead to a different successor algorithm, in which the k-subsets of $[n]$ containing n would all appear first.

3.2 Anagrams

Given letters $a_1, a_2, \ldots, a_k$ and multiplicities $n_1, n_2, \ldots, n_k \geq 0$, let $\mathcal{R}(a_1^{n_1} a_2^{n_2} \cdots a_k^{n_k})$ be the set of all rearrangements of n_1 copies of a_1, n_2 copies of a_2, and so on. It is well known that $|\mathcal{R}(a_1^{n_1} \cdots a_k^{n_k})|$ is the multinomial coefficient $\binom{n}{n_1,n_2,\ldots,n_k} = \frac{n!}{n_1!n_2!\cdots n_k!}$, where $n =$

$n_1+n_2+\cdots+n_k$. These numbers are characterized by the recursion

$$|\mathcal{R}(a_1^{n_1}\cdots a_k^{n_k})| = \sum_{\substack{i=1\\ n_i>0}}^{k} |\mathcal{R}(a_1^{n_1}\cdots a_i^{n_i-1}\cdots a_k^{n_k})|, \qquad (2)$$

when at least two n_i are positive, with initial conditions

$$|\mathcal{R}(a_1^0\cdots a_i^{n_i}\cdots a_k^0)| = 1,$$

when $n_i \geq 0$ and all other n_j are zero.

To prove (2) with the counting rules, decompose a given set $S = \mathcal{R}(a_1^{n_1}\cdots a_k^{n_k})$ into disjoint subsets S_i indexed by i's with $n_i > 0$, where S_i is the set of anagrams in S that start with the letter a_i. Deleting this initial letter gives a bijection from S_i onto the set $S_i' = \mathcal{R}(a_1^{n_1}\cdots a_i^{n_i-1}\cdots a_k^{n_k})$. Thus (2) follows from the Sum Rule and the Bijection Rule.

The corresponding successor algorithms for anagrams act as follows. For the base case where exactly one n_i is positive, the FIRST routine returns the word consisting of n_i copies of a_i, and the NEXT routine always returns LAST. Now suppose at least two n_i are positive. Define sets S, S_i, and S_i' as in the previous paragraph. FIRST(S) is the word a_iFIRST(S_i), where i is minimal with $n_i > 0$. To compute NEXT(w, S), let a_i be the first symbol in the word w. Let w' be w with this symbol erased, so $w' \in S_i'$. Return the answer a_iNEXT(w', S_i') unless the recursive call returns LAST. In that case, let j be the smallest index larger than i with $n_j > 0$. If there is no such j, return LAST; otherwise, return a_jFIRST(S_j').

The anagram successor algorithm yields as a special case a successor algorithm for permutations (take all $n_j = 1$). Similarly, using the natural bijection between k-element subsets of $[n]$ and the anagram set $\mathcal{R}(0^{n-k}1^k)$, the anagram successor algorithm also specializes to give a successor algorithm for subsets (via the Successor Bijection Rule).

3.3 Set partitions

Our next example illustrates the Successor Product Rule. Recall that a *set partition* of $[n]$ is a set of nonempty sets $P = \{B_1, B_2, \ldots, B_k\}$ such that every $j \in [n]$ belongs to exactly one of the sets B_j, which are called the *blocks* of P. Let $SP(n,k)$ be the set of set partitions of $[n]$ with k blocks. The numbers $|SP(n,k)|$ are called Stirling numbers of the second kind. These numbers are characterized by the recursion

$$|SP(n,k)| = |SP(n-1,k-1)| + k|SP(n-1,k)| \quad \text{for } 1 < k < n, \tag{3}$$

with initial conditions $|SP(n,n)| = 1$ for all $n \geq 0$ and $|SP(n,1)| = 1$ for all $n \geq 1$.

To prove (3) with the counting rules, decompose a given set $SP(n,k)$ into disjoint sets A and B, where A consists of all $P \in SP(n,k)$ such that $\{n\}$ is one of the blocks of P, and B consists of all $P \in SP(n,k)$ such that n belongs to a block of size at least two. Deleting the block $\{n\}$ leaves a set partition of $[n-1]$ into $k-1$ blocks, so we get a bijection from A onto $SP(n-1,k-1)$. On the other hand, we can define a bijection $F : B \to [k] \times SP(n-1,k)$ as follows. Suppose $P = \{B_1, B_2, \ldots, B_k\}$ is a set partition in B. We choose the indexing so that $\min(B_1) < \min(B_2) < \cdots < \min(B_k)$, where $\min(B_j)$ denotes the smallest integer in the block B_j. Now, n belongs to exactly one block of P, say B_j, where $|B_j| \geq 2$. Deleting n from its block leaves a set partition P' of $[n-1]$ into k blocks. One readily checks that the map $F(P) = (j, P')$ is invertible and hence bijective. Now (3) follows from the Bijection Rule, the Sum Rule, and the Product Rule.

We now automatically obtain successor algorithms for the sets $SP(n,k)$ by invoking the Successor Counting Rules. The one additional ingredient needed is a successor algorithm for the set $[k] = \{1, 2, \ldots, k\}$. This is straightforward: we define $\text{FIRST}([k]) = 1$, $\text{NEXT}(j, [k]) = \text{LAST}$ if $j = k$, and $\text{NEXT}(j, [k]) = j + 1$ if $j < k$.

Here is a verbal description of the recursive successor algorithm for $SP(n,k)$, where $1 < k < n$. On one hand, $\text{FIRST}(SP(n,k))$ is $\text{FIRST}(SP(n-1,k-1))$. On the other hand $\text{NEXT}(P, SP(n,k))$ is computed as follows.

Case 1: $\{n\} \in P$. In this case, let $P' = P - \{\{n\}\}$ and compute $Q' = \textsc{next}(P', SP(n-1, k-1))$. If the answer is not $\textsc{last}$, return $Q = Q' \cup \{\{n\}\}$. Otherwise, compute $Q'' = \textsc{first}(SP(n-1,k))$, and return the set partition obtained from Q'' by inserting n into the block of Q'' that contains 1.

Case 2: $\{n\} \notin P$. Label the blocks of P as described earlier, and let B_j be the block containing n. Let P' be the set partition obtained by removing n from block B_j, and recursively compute $Q' = \textsc{next}(P', SP(n-1,k))$. If Q' is not $\textsc{last}$, let the blocks of Q' be $B'_1, \ldots, B'_k$ (labeled as before), and return the set partition obtained from Q' by inserting n into block B'_j. Otherwise, let $Q'' = \textsc{first}(SP(n-1,k))$ have blocks $B''_1, \ldots, B''_k$, and (if $j < k$) return the set partition obtained by inserting n into block B''_{j+1} of Q''. If $j = k$, return $\textsc{last}$.

In this application (and in many other examples treatable by our approach), we can find a more direct description of the successor algorithm by unraveling the recursive calls. We find that the first object in $SP(n,k)$ is

$$\textsc{first}(SP(n,k))=\{\{1,\ldots,n-k+1\},\{n-k+2\},\{n-k+3\},\ldots,\{n\}\},$$

whereas the last object in $SP(n,k)$ is

$$\{\{1\},\{2\},\ldots,\{k-1\},\{k,k+1,\ldots,n-1,n\}\}.$$

To compute $\textsc{next}(P, SP(n,k))$, list all the blocks of π and erase the symbols n, $n-1$, $n-2$, one at a time, until encountering a set partition that is last in its class. When erasing each symbol, record whether that symbol was in a block by itself or (if not) the index j of the block B_j containing that symbol. Suppose the final erased symbol is m, leaving a set partition P' that is the last object in $SP(m-1,r)$.

Case 1: If m was in a block by itself just before it was erased, replace P' by $\textsc{first}(SP(m-1,r+1))$ and insert m in the lowest-indexed block of this new set partition.

Case 2: If m was in the jth block of P' just before being erased, replace P' by $\textsc{first}(SP(m-1,r))$, and insert m in block $j+1$ of the new set partition. In both cases, continue by reinserting symbols $m+1$ through n, one by one, giving them the same "block status" they had when erased. For instance, if $m+1$ was in a block by itself when it got erased, put it back in a block by itself. If $m+1$ was erased from the sth block, put it back into the sth block of the current set partition.

Like the Successor Sum Rule, the Successor Product Rule is "non-commutative." For example, consider the recursion

$$|SP(n,k)| = |SP(n-1,k-1)|+|SP(n-1,k)|k, \text{ for } 1<k<n, \quad (4)$$

which is obtained from (3) by changing the term $k|SP(n-1,k)|$ to $|SP(n-1,k)|k$. This change leads to a different (and probably more efficient) successor algorithm. Specifically, referring to Case 2 of the recursive description of $\textsc{next}(P, SP(n,k))$ above, the new subroutine usually acts by moving the value n from block B_j to B_{j+1} of P. We only need to make the recursive call $Q' = \textsc{next}(P', SP(n-1,k))$ when n starts in block B_k. In this situation, if Q' is not $\textsc{last}$, we obtain the answer by inserting n into block B'_1 of Q'; otherwise we return $\textsc{last}$.

3.4 Derangements

A *derangement* of $[n]$ is a bijection $f:[n]\to[n]$ such that $f(i)\neq i$ for all $i\in[n]$. Let $D(n)$ be the set of all derangements of $[n]$. The numbers $|D(n)|$ are characterized by the recursion

$$|D(n)| = (n-1)|D(n-1)| + (n-1)|D(n-2)| \text{ for all } n\geq 2, \quad (5)$$

with initial conditions $|D(0)|=1$ and $|D(1)|=0$.

To prove (5) using the counting rules, we view each function $f:[n]\to[n]$ as a directed graph with vertex set $[n]$ and directed edge set $\{(i,f(i)) : i\in[n]\}$. It is well known that f is a bijection iff its directed graph is a disjoint union of directed cycles. Moreover,

f is a derangement iff all the cycles in its directed graph have length greater than 1. For fixed $n \geq 2$, write $D(n)$ as the disjoint union $A \cup B$, where A is the set of $f \in D(n)$ in which n belongs to a cycle of length 3 or more, and B is the set of $f \in D(n)$ in which f belongs to a cycle of length 2. Given $f \in A$, we obtain a derangement $f' \in D(n-1)$ by removing n from its cycle. More specifically, if $f(a) = n$ and $f(n) = b$, we define $f'(a) = b$ and $f'(j) = f(j)$ for all $j \neq a$ in $[n-1]$. The passage from f to f' is reversible if we remember the value b following n in its cycle. Thus, the map sending f to the pair (b, f') is a bijection $G : A \to [n-1] \times D(n-1)$. Similarly, given $f \in B$, we obtain a derangement $f'' \in D(n-2)$ by removing the cycle of length 2 containing n and renumbering the remaining vertices. More specifically, if $f(n) = c$ and $f(c) = n$, the directed graph of f'' is obtained by erasing the cycle (n, c) and then subtracting 1 from all remaining vertex labels that exceed c. The map sending f to the pair (c, f'') is a bijection $H : B \to [n-1] \times D(n-2)$. Now (5) follows from the Bijection Rule, the Sum Rule, and the Product Rule.

This counting argument translates into the following successor algorithm for derangements. For $n \geq 3$, we find $f = \text{FIRST}(D(n))$ by initially computing $f' = \text{FIRST}(D(n-1))$, and then splicing n into a cycle just before 1. For $n = 2$, $\text{FIRST}(D(2))$ is the 2-cycle $(1, 2)$. Unraveling the recursion, we see that for all $n \geq 2$, $\text{FIRST}(D(n))$ is the n-cycle $(1, 2, 3, \ldots, n)$. To compute $\text{NEXT}(f, D(n))$ for $n \geq 4$, first suppose $f \in A$ with $G(f) = (b, f')$. Return $G^{-1}(b, \text{NEXT}(f', D(n-1)))$ if the recursive call to NEXT does not yield value LAST. Otherwise, if $b < n-1$, return $G^{-1}(b+1, \text{FIRST}(D(n-1)))$. Otherwise, return $H^{-1}(1, \text{FIRST}(D(n-1)))$. Next suppose $f \in B$ with $H(f) = (c, f'')$. Return $H^{-1}(c, \text{NEXT}(f'', D(n-1)))$ if the recursive call to NEXT does not yield value LAST. Otherwise, if $c < n-1$, return $H^{-1}(c+1, \text{FIRST}(D(n-1)))$. Otherwise, return LAST. (For $n = 2$ or $n = 3$, a few modifications are needed since $D(1) = \emptyset$.)

As with set partitions, we can build a different successor algorithm for derangements that makes fewer recursive calls by writing the recursion (5) in the form

$$|D(n)| = |D(n-1)| \cdot (n-1) + |D(n-2)| \cdot (n-1) \quad \text{for all } n \geq 2. \quad (6)$$

3.5 Other families of objects

The method of converting counting arguments to successor algorithms applies to many families of combinatorial objects beyond the ones considered in the preceding examples. As a sample, we mention the following structures.

- *Words.* A word of length m in a k-letter alphabet can be identified with an m-tuple in the product set $[k]^m$ by numbering the letters of the alphabet $1, 2, \ldots, k$ in some fashion. A successor algorithm for these words arises immediately by combining the Successor Product Rule with the simple successor subroutine for the set $[k]$ described in §3.3. This algorithm essentially implements counting in base k.

- *Permutations.* We have already mentioned that permutations are a special case of anagrams (see §3.2). Alternatively, let Perm(n) be the set of permutations of $[n]$. We get a bijection from Perm(n) to $[n] \times \text{Perm}(n-1)$ by removing the value n from a permutation of $[n]$ and remembering which position was occupied by n. A successor algorithm for Perm(n) now follows from the Successor Bijection Rule and the Successor Product Rule.

- *Set partitions of any size.* Let $SP(n)$ be the set of all set partitions of $[n]$ with any number of blocks. The numbers $|SP(n)|$ are called Bell numbers. On one hand, $SP(n)$ is clearly the disjoint union of the sets $SP(n,k)$ as k ranges from 1 to n. So one successor algorithm for $SP(n)$ follows by combining successor routines for $SP(n,k)$ using the Successor Sum Rule. Alternatively, the Bell numbers are known to satisfy the recursion

$$|SP(n)| = \sum_{k=0}^{n-1} \binom{n-1}{k} |SP(n-1-k)| \quad \text{for all } n \geq 1, \tag{7}$$

  with initial condition $|SP(0)| = 1$. We can prove this recursion by an explicit counting argument (the idea is to remove n and all elements in its block from a set partition of $[n]$), so we obtain another successor algorithm for $SP(n)$.

- *Integer partitions.* An integer partition of n is a weakly decreasing list of positive integers (called *parts*) whose sum is n. Let $IP(n,k)$ be the set of all integer partitions of n into k parts. One readily proves the recursion

$$|IP(n,k)| = |IP(n-1,k-1)| + |IP(n-k,k)|, \qquad (8)$$

  for $0 < k \leq n$ by a counting argument. The Successor Sum Rule and Successor Bijection Rule automatically provide a successor algorithm for the sets $IP(n,k)$.

- *Catalan structures.* The Catalan numbers $C_n = \frac{1}{n+1}\binom{2n}{n}$ count many collections of combinatorial objects, including *Dyck paths* of order n. These are paths consisting of n north steps and n east steps from $(0,0)$ to (n,n) such that the path never goes below the line $y = x$. One readily proves the Catalan recursion

$$C_n = \sum_{k=1}^{n} C_{k-1}C_{n-k}, \quad \text{for all } n > 0 \qquad (9)$$

  by considering the least integer $k > 0$ where a Dyck path touches a point (k,k) on the diagonal. The Successor Rules convert the proof of this recursion to a successor algorithm for Dyck paths. Using appropriate bijections, one obtains successor algorithms for many other structures enumerated by Catalan numbers.

- *Trees.* An n-vertex labeled tree is a connected simple undirected graph on the vertex set $[n]$ having no cycles. The number of such trees is $[n]^{n-2}$. Eğecioğlu and Remmel [2] defined a bijection from the set of n-vertex labeled trees onto the set of words $w \in [n]^n$ with $w_1 = 1$ and $w_n = n$. Using the Successor Bijection Rule and successor subroutines for words, we thereby obtain successor algorithms for trees. We can also restrict attention to trees with specified degree sequences, by using Prüfer codes [6] to map these trees bijectively to a set of anagrams determined by the degree sequence.

- *Tableaux.* Given an integer partition λ of n, the diagram of λ is an array of n boxes, with λ_i left-justified boxes in the

ith row from the bottom. A standard tableau of shape λ is a filling of the diagram of λ with the integers 1 through n, used once each, such that the values in all rows increase reading left to right, and the values in all columns increase reading bottom to top. Let $\mathrm{SYT}(\lambda)$ be the set of standard tableaux of shape λ. By removing n from a standard tableau of shape λ, one can prove the recursion

$$|\,\mathrm{SYT}(\lambda)| = \sum_{\mu} |\,\mathrm{SYT}(\mu)|, \tag{10}$$

summed over all μ arising from λ by removing a corner cell.

We thereby obtain a successor algorithm for the sets $\mathrm{SYT}(\lambda)$ via the Successor Sum Rule and Successor Bijection Rule. Here we must specify a total ordering of the partitions μ arising from λ, say based on a left-to-right ordering of the corner cells removed from λ to produce μ.

This list of examples is far from exhaustive, but it should convince the reader of the wide applicability of the methods presented here. One limitation of the Successor Rules is that they do not automatically apply to counting arguments relying on indirect methods, such as generating functions or algebraic simplifications involving subtraction or division. Moreover, there often exist more efficient successor algorithms than the ones produced by the Successor Rules (e.g., loopless subroutines or Gray codes). Many such algorithms are discussed in detail in [3, Sec. 7.2].

4 Comparison to other approaches

This section briefly compares our Successor Rules to two other general approaches for creating successor algorithms: decision trees and the "combinatorial family" concept of Nijenhuis and Wilf. The main benefit of our method is its extreme simplicity: counting arguments based on the Bijection Rule, Sum Rule, and Product Rule can be converted automatically and effortlessly into successor algorithms. There is no need to worry about auxiliary data structures or complicated mathematical formalisms.

4.1 Decision trees

One common method for finding successor algorithms is the decision tree paradigm (see, for instance, the textbook of Bender and Williamson [1]). We describe decision trees in the context of the specific example of enumerating set partitions of n into k blocks (see §3.3). We rewrite the recursion (3) as a sum of $k+1$ terms:

$$\begin{aligned}|SP(n,k)| = |SP(n-1,k-1)| + |SP(n-1,k)| \\ + |SP(n-1,k)| + \cdots + |SP(n-1,k)|. \qquad (11)\end{aligned}$$

We imagine creating a set partition $P \in SP(n,k)$ by making a sequence of decisions. Each internal node in the tree represents a particular decision we make in the process of building P. At the root node of the tree, we choose from the following ordered list of $k+1$ alternatives: should n be in a block by itself, or in block B_1, or in block B_2, ..., or in block B_k? (Here the blocks B_j will have other elements besides n and are indexed by increasing minimum element, as in §3.3.) The root node has an ordered list of $k+1$ edges leading down to the next level, corresponding to the $k+1$ choices we could make. If we follow the leftmost edge, we arrive at a node where we make an analogous choice for $n-1$ relative to the set $SP(n-1,k-1)$. Similarly, the other k edges lead to nodes where we make decisions based on the recursion for $|SP(n-1,k)|$. The tree keeps growing until we reach an initial condition for the recursion, which produces a leaf node containing a single set partition.

After the full decision tree is built, each set partition in $SP(n,k)$ appears in exactly one leaf of the tree. Using standard algorithms for traversing ordered trees, we can go from a particular leaf in the tree to the "next" leaf (in a left to right scan of the leaves) by moving up and down the edges of the decision tree. For many combinatorial objects of interest (such as permutations, subsets, etc.), one can often unravel the tree traversal process to obtain a more direct description of the successor algorithm that does not specifically mention the tree.

It is not hard to see that the decision tree technique, when applied to a recursion that can be written as a sum of terms, produces the same results as our Successor Sum Rule. Our method is simpler in the sense that the programmer need not explicitly design, build, or traverse the decision tree. Indeed, the recursive calls to NEXT carry out these bookkeeping tasks implicitly and automatically.

4.2 Combinatorial families

In [5, Chpt. 13, pp. 100–102], Nijenhuis and Wilf define the concept of a *combinatorial family* and describe a NEXT subroutine for listing objects in such a family. The combinatorial family itself is a directed graph G whose vertex set $V(G)$ is partially ordered with a unique minimal element τ, such that each edge of G goes from a vertex v to a smaller vertex w in the partial ordering. Moreover, every vertex except τ has at least one outgoing edge, and there is a total ordering on the set of edges leaving any given vertex v. Multiple edges can go from v to w, but the graph must be locally finite. A *combinatorial object* in the family is a directed walk in G from some vertex v to τ.

The successor algorithm for finding the next object after a given walk W proceeds as follows. Start at τ and backtrack along the walk until first reaching an outgoing edge e that is not the final edge departing from its initial vertex v. If no such e exists, W is the last walk. Otherwise, replace e by the next edge in the ordered list of edges leaving v, and then complete the new walk by repeatedly following the first edge from each new vertex until reaching τ.

We claim that this successor algorithm for combinatorial families can be realized as a special case of the Successor Sum Rule. For each vertex v, let $W(v)$ be the set of directed walks from v to τ. Let the edges leaving v be $e_1, \ldots, e_k$ in this order. For $1 \leq i \leq k$, let $W_i(v)$ be the set of walks in $W(v)$ that begin by following the edge e_i. Evidently, $W(v)$ is the disjoint union of the sets $W_i(v)$, so $|W(v)| = \sum_{i=1}^{k} |W_i(v)|$. It is routine to check that the successor algorithm described in the previous paragraph is exactly the algorithm produced by applying the Successor Sum Rule to these recursions characterizing the numbers $|W(v)|$. The initial condition is $|W(\tau)| = 1$.

To use the Nijenhuis-Wilf approach to obtain successor algorithms for objects such as subsets, permutations, set partitions, etc., one must first find an "encoding bijection" mapping such objects to directed walks in an appropriate directed graph satisfying the definition of a combinatorial family. Examples of such encodings are given in [5, pp. 103–105]. The benefits of our approach using Successor Counting Rules are similar to those cited for decision trees: we need not explicitly construct the directed graphs or the walks, nor is it needed for the programmer to find encodings of the objects or backtrack through walks in the graph. Of course, objects are still encoded implicitly in our method through the use of the Successor Bijection Rule, but any necessary manipulations of the objects are already present in the original counting argument.

5 Rules for ranking and unranking

Given an n-element set S, a *ranking procedure* for S is an algorithm implementing a bijection $F : S \to [n]$, where $[n]$ is the set $\{1, 2, \ldots, n\}$. The corresponding *unranking procedure* for S is an algorithm implementing the inverse bijection $G : [n] \to S$. Intuitively, $G(1), G(2), \ldots, G(n)$ is a list of all elements of S in a particular order. Given any object $x \in S$, $F(x)$ gives the position (or "rank") of x in this list without requiring us to generate the entire list. Similarly, we can determine the object occupying position j on the list by computing $G(j)$. Among other applications, unranking algorithms allow us to generate random objects in the set S. To do so, we use any standard random sampling algorithm to pick an integer j from the set $[n]$, then return the object $G(j)$.

Earlier, we converted the Bijection Rule, Sum Rule, and Product Rule into successor rules for building successor algorithms. Here, we describe versions of these three rules for building ranking and unranking algorithms. The new rules apply to any set S that we can count using the three fundamental counting principles. As before, the ranking and unranking rules automatically convert any counting argument for S into explicit ranking and unranking algorithms for S. In particular, we can obtain such algorithms for all the examples considered earlier.

Here are the ranking versions of the three basic counting rules.

Ranking Bijection Rule. Let $H : S \to T$ be a bijection with inverse $H^{-1} : T \to S$, and suppose we already know a ranking bijection $F : S \to [n]$ and the associated unranking bijection $G : [n] \to S$. Then a ranking bijection for T is the composition $F \circ H^{-1} : T \to [n]$. The associated unranking bijection for T is the composition $H \circ G : [n] \to T$.

The validity of this rule follows at once from the fact that the composition of bijections is a bijection.

Ranking Sum Rule. Let $(S_1, S_2, \ldots, S_k)$ be a given sequence of nonempty pairwise disjoint finite sets such that $|S_i| = n_i$ and we already know ranking algorithms $F_i : S_i \to [n_i]$ and associated unranking algorithms $G_i : [n_i] \to S_i$ for $i = 1, 2, \ldots, k$. Define $S = S_1 \cup S_2 \cup \cdots \cup S_k$ and $n = n_1 + n_2 + \cdots + n_k$. We obtain a ranking bijection $F : S \to [n]$ as follows. Given $x \in S$, find the unique i with $x \in S_i$, and define

$$F(x) = n_1 + n_2 + \cdots + n_{i-1} + F_i(x).$$

The associated unranking bijection $G : [n] \to S$ acts as follows. Given $j \in [n]$, find the unique i such that $n_1 + n_2 + \cdots + n_{i-1} < j \leq n_1 + n_2 + \cdots + n_i$, and define

$$G(j) = G_i(j - [n_1 + n_2 + \cdots + n_{i-1}]).$$

Just like the Successor Sum Rule, the Ranking Sum Rule lists the elements of S by concatenating lists for $S_1, S_2, \ldots, S_k$ in this order (as shown in Figure 1). Given $x \in S_i$, we find the position $F(x)$ of x in the list for S by moving past the $n_1 + \cdots + n_{i-1}$ positions occupied by the elements in $S_1, \ldots, S_{i-1}$, then finding the position of x in the list for S_i. The recipe for G inverts this process, as is readily verified.

Ranking Product Rule. Let $S_1, S_2, \ldots, S_k$ be given nonempty finite sets such that $|S_i| = n_i$ and we already know ranking algorithms $F_i : S_i \to [n_i]$ and associated unranking algorithms $G_i : [n_i] \to S_i$ for $i = 1, 2, \ldots, k$. Define $S = S_1 \times S_2 \times \cdots \times S_k$ and $n = n_1 \cdot n_2 \cdot \ldots \cdot n_k$. We obtain a ranking bijection $F : S \to [n]$ as follows. Given $x = (x_1, x_2, \ldots, x_k) \in S$, first compute $(j_1, j_2, \ldots, j_k)$ where $j_i = F_i(x_i)$ for $i = 1, 2, \ldots, k$. Then define

$$\begin{aligned} F(x) &= (j_1 - 1)n_2 n_3 \cdots n_k \\ &\qquad + (j_2 - 1)n_3 \cdots n_k + (j_3 - 1)n_4 \cdots n_k \\ &\qquad + \cdots + (j_{k-1} - 1)n_k + j_k \\ &= 1 + \sum_{i=1}^{k} (j_i - 1) \prod_{s=i+1}^{k} n_s. \end{aligned}$$

The associated unranking bijection $G : [n] \to S$ acts as follows. Given $j \in [n]$, we first find $(j_1, j_2, \ldots, j_k)$ with each $j_i \in [n_i]$ as follows. Divide $j - 1$ by $d_1 = n_2 n_3 \cdots n_k$ to get a quotient q_1 and remainder r_1 with $0 \le r_1 < d_1$; let $j_1 = q_1 + 1$. Then divide r_1 by $d_2 = n_3 \cdots n_k$ to get a quotient q_2 and remainder r_2 with $0 \le r_2 < d_2$; let $j_2 = q_2 + 1$. Continue similarly to find $j_3, \ldots, j_k$; finally, return $G(j) = (G_1(j_1), G_2(j_2), \ldots, G_k(j_k))$.

The formulas for F and G in the Ranking Product Rule may appear unwieldy, but (as in the case of the Successor Product Rule) these formulas emerge automatically from previous simpler rules. In particular, the Ranking Product Rule for $k = 2$ sets arises from the Ranking Sum Rule via the same method discussed at the end of Section 2. In more detail, let us write $S_1 = \{s_1, \ldots, s_{n_1}\}$ and $S_2 = \{t_1, \ldots, t_{n_2}\}$ where $s_{j_1} = G_1(j_1)$ for $j_1 = 1, 2, \ldots, n_1$ and $t_{j_2} = G_2(j_2)$ for $j_2 = 1, 2, \ldots, n_2$. Think of $S = S_1 \times S_2$ as the disjoint union of the sets

$$\{s_1\} \times S_2, \{s_2\} \times S_2, \ldots, \{s_{n_1}\} \times S_2$$

in this order. Each of these sets has size n_2 by the Bijection Rule. Using the Ranking Sum Rule to compute $F(s_{j_1}, s_{j_2})$, we skip over the first $j_1 - 1$ sets and find s_{j_2} in the j_2'th position in the set $\{s_{j_1}\} \times S_2$. Thus, $F(s_{j_1}, s_{j_2}) = (j_1 - 1)n_2 + j_2$, in agreement with the formula for F in the Ranking Product Rule when $k = 2$. If the output of F is the integer $j \in [n_1 n_2]$, we recognize $j_1 - 1$ and $j_2 - 1$ as the unique quotient and remainder when $j - 1$ is divided by n_2. This explains the formula for $G = F^{-1}$ in the Ranking Product Rule when $k = 2$.

Finally, the Ranking Product Rule for $k + 1$ sets follows by applying the version of this rule for two sets to the Cartesian product $S_1 \times (S_2 \times \cdots \times S_{k+1})$, since we can assume by induction that a ranking map of the required form has already been built for the k-fold Cartesian product $S_2 \times \cdots \times S_{k+1}$. It is routine to unravel this recursive construction to get the formulas for F and G stated above. Moreover, as one readily checks, the total ordering of S determined by the Ranking Rules in this section coincides with the total ordering of S built via the Successor Rules in Section 2.

As a final remark, we mention an open problem concerning the Successor Rules. Can these rules (or similar techniques) provide a uniform framework for automatically developing minimal-change listings such as Gray codes or de Bruijn sequences? The author's initial investigation of small examples suggests this may be a subtle question. One difficulty is that the links between the last element in set S_i and the first element in set S_{i+1} (see Figure 1) do not necessarily correspond to allowable changes in the underlying objects, even if we already have minimal-change listings for the individual sets S_i. I hope to return to this question in future work.

Acknowledgements

I thank the anonymous referee for helpful suggestions, including the open question raised in the previous paragraph. This work was partially supported by a grant from the Simons Foundation (#244398 to Nicholas Loehr).

References

[1] E. Bender and S.G. Williamson, *Foundations of Combinatorics with Applications*, Dover, Mineola, NY (2006).

[2] Ö. Eğecioğlu and J. Remmel, "Bijections for Cayley trees, spanning trees, and their q-analogues," *J. Combin. Theory Ser. A* **42** (1986), 15–30.

[3] D. Knuth, *The Art of Computer Programming*, Volume 4 (multiple fascicles), Addison-Wesley, Reading, MA (2005).

[4] D.L. Kreher and D.R. Stinson, *Combinatorial Algorithms: Generation, Enumeration, and Search*, CRC Press (1999).

[5] A. Nijenhuis and H. Wilf, *Combinatorial Algorithms*, Academic Press, New York, NY (1975).

[6] H. Prüfer, "Neuer Beweis eines Satzes über Permutationen," *Arch. Math. Phys.* **27** (1918), 742–744.

[7] E. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ (1977).

[8] F. Ruskey, *Combinatorial Generation* (preliminary draft).

[9] D. Stanton and D. White, *Constructive Combinatorics*, Springer-Verlag, New York, NY (1986).

[10] H. Wilf, *Combinatorial Algorithms: An Update*, SIAM (1989).