Uncovering the Structural Geometry of Permutations: An Expository Framework

George W. Taylor
gtaylor@tropicalcoder.com
10.5281/zenodo.16761201

August 7, 2025

Abstract

This paper introduces a recursive framework for lexicographic permutations, revealing inherent structural symmetries such as midpoint mirroring, prefix nesting, and factorial block decomposition. This structure yields a global reflection symmetry, allowing mirrored permutations to be computed with negligible additional cost. It also supports compressed codes that encode each permutation's offset from the lexicographic minimum. We present a recursive algorithm for permutation generation that exposes this internal structure, along with a variant for direct compressed code production. We compare with classical methods and include implementations in Python and C++.

1. Introduction

While lexicographic order is well understood, its internal architecture—particularly the symmetries revealed in delta sequences—has not been widely explored compositionally. This paper uncovers a compact recursive pattern embedded within permutation space, reframing lexicographic order as self-similar through the lens of delta structure. It introduces a mirrored embedding and recursive organization of delta sequences across factorial boundaries.

These insights enable efficient representation, indexing, and traversal, including reflection-based computation of permutation pairs at negligible additional cost. Collectively, they support reinterpretation of known results from a structural perspective, offering new opportunities for analysis and application. Building on these insights, we define a compact numerical representation—the *compressed code*—which arithmetically encodes each permutation's offset from the lexicographic minimum. These codes support direct reconstruction and extend naturally to other numeral systems.

Classical permutation algorithms such as Heap's and Johnson-Trotter emphasize efficiency and minimal-change transitions. These state-driven procedures treat permutation space atomistically, lacking a global structural interpretation. While Lehmer unranking reveals the factorial organization of permutations procedurally, it does not emphasize the recursive symmetries and alignments explored here. Although recursive generation algorithms for lexicographic order exist in textbooks, their extension towards structural compression remains largely unexplored. Similarly, the factorial number system provides positional indexing but lacks the recursive compression and symmetry exploitation emphasized here.

In 2011, the author (Taylor) proposed a model of permutation space in "A Permutation on Combinatorial Algorithms", and experimented with *structurally aware* generation. This work described recursive sym-

metries across digit lengths and suggested encoding permutations by their offset from the lexicographic minimum. It concluded with a conjecture questioning whether such codes could be generated directly, without enumeration of the permutations themselves.

Building on those foundations, we formalize and expand upon the structural properties originally observed, adding a recursive tree whose branching patterns reflect prefix grouping and digit relabeling (see Appendix G). We present a recursive algorithm whose structure directly embeds this tree and a variation that outputs compressed codes, providing implementations that emphasize conceptual clarity. In this paper, we treat permutations as beginning with 0 and express them as undelimited digit sequences (base-10 assumed).

2. Recursive Symmetries in Delta Space

The symmetry of the permutation space is not evident in the sequence of values, but rather in the **differences between them**—the deltas. This derivative reveals a hidden order not apparent in the raw permutation values.

Let each permutation be interpreted as a base-10 integer. We define the delta as the numeric difference between successive permutations P in lexicographic order:

$$\Delta_i = P_{i+1} - P_i$$

Delta Table for Lexicographic Permutations of 4 Digits:

Index	Delta Expression	
0:	$0132 - 0123 = 9$	
1:	$0213 - 0132 = 81$	
2:	$0231 - 0213 = 18$	
3:	$0312 - 0231 = 81$	
4:	$0321 - 0312 = 9$	
5:	$1023 - 0321 = 702$	
6:	$1032 - 1023 = 9$	
7:	$1203 - 1032 = 171$	
8:	$1230 - 1203 = 27$	
9:	$1302 - 1230 = 72$	
10:	$1320 - 1302 = 18$	
11:	$2013 - 1320 = 693$	<-- centre of symmetry (pivot)
12:	$2031 - 2013 = 18$	
13:	$2103 - 2031 = 72$	
14:	$2130 - 2103 = 27$	
15:	$2301 - 2130 = 171$	
16:	$2310 - 2301 = 9$	
17:	$3012 - 2310 = 702$	
18:	$3021 - 3012 = 9$	
19:	$3102 - 3021 = 81$	
20:	$3120 - 3102 = 18$	
21:	$3201 - 3120 = 81$	
22:	$3210 - 3201 = 9$	

The delta list for the complete 4-digit permutation space reveals three distinct segments:

- The first five deltas reproduce the delta sequence of the 3-digit permutations.
- The central 13 deltas are unique to the 4-digit set and centre around the pivot permutation.
- The final five deltas mirror the first five—forming a perfect reflection.

Given the pivot at delta index $\frac{n!-2}{2}$ (0-based, as in the delta table), the deltas satisfy the following identity:

$$\Delta_{\text{pivot}+d} = \Delta_{\text{pivot}-d}$$

for all valid integer offsets d. This relation expresses exact equality and confirms the palindromic symmetry of the delta sequence. It arises directly from the recursive embedding and central reflection. Its implications for the permutations themselves will be explored in the sections that follows.

These observations reveal two intertwined symmetries in what we call *delta space*:

- **Recursive (structural) symmetry**: The delta sequences at level n are built by embedding palindromic copies of level $n-1$, creating self-similar recursive layers.
- **Arithmetic (delta-based) symmetry**: The delta table shows arithmetic reflection around a central pivot, from which the formula above emerges.

3. Why Delta Sequences Recursively Embed Lower-Order Structures

The appearance of Δ_k patterns within Δ_n arises naturally from how lexicographic permutations are constructed. As we build the list, we begin by permuting just the final two digits, then expand to include the last three, then four, and so on—progressively increasing the digit count of the active suffix up to $n-1$.

In each of these regions, the digits to the left of the permuted block remain fixed. When computing the delta between successive permutations, these fixed digits cancel out, leaving only the contribution of the changing suffix. As a result, each local delta pattern reflects the same differences we would observe in a smaller permutation set: Δ_2, Δ_3, Δ_4, and so on, until finally all digits are in motion.

In the second half of the list, the process unwinds: as each leading digit settles into place, the active suffix shrinks by one digit at a time. The deltas mirror the same patterns—first those of Δ_{n-1}, then smaller Δ_k sequences—ultimately converging back toward Δ_2.

This gradual buildup and teardown of the active suffix defines the structure of the prefix and suffix regions of Δ_n. Though not directly visible in the lexicographic ordering itself, this progression governs the shape of the delta sequence, which inherits this symmetry as a natural palindromic reflection.

4. Recursive Construction of Delta Layers

To illustrate the recursive structure more explicitly, we now introduce notation that will help express it.

Let Δ_n denote the delta sequence of length $n! - 1$ associated with the full permutation set S_n.

We define the *central spline* C_n as the core of Δ_n, bridging two mirrored copies of Δ_{n-1} across the midpoint, arising from transitions between major prefix changes in the lexicographic generation process. It is palindromic in its own right.

We express this symbolically:

- Δ_2: Base sequence $[9]$ for $\{0, 1\}$.
- $C_3, C_4, \ldots$: Central splines (e.g., $C_3 = [81, 18, 81]$).

This recursive embedding can be seen explicitly in the symbolic expansion of Δ_n for

$n = 4$:

$$[\Delta_2, C_3, \Delta_2, C_4, \Delta_2, C_3, \Delta_2]$$

$n = 5$:

$$[\Delta_2, C_3, \Delta_2, C_4, \Delta_2, C_3, \Delta_2, C_5, \Delta_2, C_3, \Delta_2, C_4, \Delta_2, C_3, \Delta_2]$$

This embedding process may be formalized as:

$$\Delta_n = \Delta_{n-1} \parallel C_n \parallel \Delta_{n-1}$$

Each layer recursively embeds lower-order segments and central splines in a palindromic pattern, matching the structure observed in the full delta table. Although we do not formalize the spline values here, this layered recurrence captures the recursive and palindromic structure observed in the delta table.

Having traced the recursive structure of delta space, we now turn our attention to permutation space.

5. Arithmetic Reflection in Permutation Space

As shown in Section 2, delta space exhibits mirror symmetry centered on a pivot permutation. A corresponding symmetry exists in permutation space, where the lexicographically ordered sequence reflects about a numerical midpoint. This midpoint is defined as:

$$\text{midpoint} = \frac{P_{\text{max}} + P_{\text{min}}}{2}$$

where P_{min} and P_{max} are the integer values of the first and last permutations. While this midpoint defines the axis of symmetry, the reflected permutation is more efficiently computed without it, using:

$$P_{\text{reflected}} = P_{\text{max}} - (P_i - P_{\text{min}})$$

Example: With $n = 4$, there are $4! = 24$ permutations, ranging from $P_{\text{min}} = 0123 = 123$ to $P_{\text{max}} = 3210$. Given the permutation $P_i = 2301$, its reflected counterpart is computed by subtracting the offset from the maximum:

$$P_{\text{reflected}} = P_{\text{max}} - (P_i - P_{\text{min}}) = 3210 - (2301 - 123) = 1032$$

The reflection allows a second permutation to be obtained at negligible additional cost, enabling efficient traversal of the permutation space. This formula is algorithm-independent, requiring only the numeric permutation value and the known bounds of the permutation space. (Lexicographic order can be recovered via buffering if desired.)

For algorithms that track permutation indices, an analogous reflection applies: The 0-based reflected index is given by:

$$i_{\text{reflected}} = (n! - 1) - i$$

Example: If the permutation $P_i = 2301$ occurs at index $i = 16$, and indices range from 0 through 23, then:

$$i_{\text{reflected}} = 23 - 16 = 7$$

Permutation 1032 lies at index 7, flipping the index around the midpoint. While index reflection follows from a general arithmetic symmetry over consecutive integers, permutation reflection leverages the inherent structural symmetry of permutation space.

6. Structural Embedding in Permutation Space

The recursive embedding described in Section 2 for delta sequences appears to manifest within the permutations themselves, though obscured by lexicographic ordering. In particular, the first and last $(n-1)!$ permutations of S_n seem to echo the structure of S_{n-1}.

- In the **last** $(n-1)!$ permutations of S_n, remove the first digit from each. The remaining subsequences form the complete set of permutations for S_{n-1}.
- In the **first** $(n-1)!$ permutations of S_n, remove the first digit and subtract 1 from each of the remaining digits. This again yields the complete set of permutations for S_{n-1}.

For example, let us look at the first and last permutations for set $n = 4$:

Last $(n-1)!$ **permutations**		**First** $(n-1)!$ **permutations**	
First digit removed $\rightarrow S_3$		**First digit removed, subtract 1** $\rightarrow S_3$	
3 0 1 2	0 1 2	0 1 2 3	0 1 2
3 0 2 1	0 2 1	0 1 3 2	0 2 1
3 1 0 2	1 0 2	0 2 1 3	1 0 2
3 1 2 0	1 2 0	0 2 3 1	1 2 0
3 2 0 1	2 0 1	0 3 1 2	2 0 1
3 2 1 0	2 1 0	0 3 2 1	2 1 0

Table 1: Recursive embedding visible directly in the permutations for $n = 4$.

Just as the first and last $(n-1)!$ permutations of S_n embed the structure of S_{n-1}, this relationship can be extended to construct extremal regions of S_n given the full set S_{n-1}. Specifically:

- To generate the **first** $(n-1)!$ permutations of S_n, increment every digit of each permutation in S_{n-1} by 1, then prepend 0.
- To generate the **last** $(n-1)!$ permutations of S_n, prepend the maximum digit $n-1$ to each permutation in S_{n-1} without modifying the digits.

What initially appeared as structural reflection turns out to follow a consistent numerical rule across the entire permutation list, leading naturally to a different kind of recursion—one rooted in a systematic partitioning of permutation space by leading digit. This marks the shift from structural reflection to combinatorial encoding, forming the basis for ranking, unranking, and the Lehmer code itself.

7. Factorial Partitioning of Permutation Space

Any list of $n!$ permutations can be partitioned into n consecutive groups of $(n-1)!$, reflecting a fundamental property of factorials: $n!$ is evenly divisible by $(n-1)!$.

Example ($n = 4$). The lexicographic list has $4! = 24$ permutations that can be partitioned into four groups of $3! = 6$ each.

$$24 \text{ permutations } = \underbrace{6 \ + \ 6 \ + \ 6 \ + \ 6}_{\text{groups of } 6}$$

This block structure reveals a different kind of order—not from mirrored patterns in delta space, but from the arithmetic of permutation indexing itself. Indeed, the permutations of $\{0, 1, \ldots, n-2\}$ can be recovered from *every* group, a known property fundamental to Lehmer codes. (See Appendix F for empirical observations related to this structural regularity.)

There is one uniform rule that specializes to the first and last cases discussed above in Section 6. Each group is determined by a fixed leading digit $k \in \{0, 1, \ldots, n-1\}$ (assuming lexicographic order).

Uniform rule for any group

(a) Remove the leading digit k.

(b) In the resulting $(n-1)$-digit string, subtract 1 from every digit that is strictly greater than k.

This mapping sends the digits in $\{0, 1, \ldots, n-1\} \setminus \{k\}$ onto $\{0, 1, \ldots, n-2\}$ in an order-preserving way, thereby producing each permutation of S_{n-1}.

Special cases:

- For the **first** group ($k = 0$): Every remaining digit exceeds 0, so subtract 1 from all, recovering S_{n-1}.
- For the **last** group ($k = n-1$): No remaining digits exceed k, so no changes occur after removal, again recovering S_{n-1}.

Conversely, we can construct each group from S_{n-1}:

- For leading digit k: Take each permutation in S_{n-1}, add 1 to every digit that is $\geq k$, and prepend k.

Applying this to each k reconstructs the full set S_n.

Up to this point, we have only used the uniform rule to reveal S_{n-1} inside S_n. But the same rule applies again inside S_{n-1}, revealing S_{n-2}; and inside that, S_{n-3}, and so on. This cascading structure means that any permutation of S_n can be dissected step by step to smaller sets until nothing remains. In the next section, we follow this chain of reductions for a single permutation—a process we call *decomposition*.

8. Decomposition and Reconstruction of a Permutation

To see how this process unfolds in practice, we apply the uniform rule to recursively decompose a permutation, step by step.

Consider the permutation 2310 (for $n = 4$).

- 2310: Leading digit $k = 2$; remove it $\rightarrow$ 310; subtract 1 from any digit greater than 2 $\rightarrow$ 210.
- 210: Leading digit $k = 2$; remove it $\rightarrow$ 10; no digits greater than 2, so unchanged.
- 10: Leading digit $k = 1$; remove it $\rightarrow$ 0; no digits greater than 1, so unchanged.
- 0: Base case.

This sequence of reductions corresponds to moving down a *factorial tree* of permutation space, where each level branches into n subtrees, each isomorphic to the tree for S_{n-1} (see Appendix G for an illustration).

The sequence of removed digits is $2, 2, 1$, followed by the final 0. These values record the choices made at each level: with m digits remaining, the number gives the index of the selected digit from the sorted remaining list $\{0, 1, \ldots, m-1\}$, where m decreases from 4 to 1.

Reconstruction reverses the process: start from an empty sequence and insert digits in order $0, 1, 2, 3$, placing each according to the reversed choice sequence $[2, 2, 1, 0]$:

- Start with an empty sequence; insert 0 at position 0 $\rightarrow$ 0.
- Insert 1 at position 1 $\rightarrow$ 10.
- Insert 2 at position 2 $\rightarrow$ 210.
- Insert 3 at position 2 $\rightarrow$ 2310.

This choice sequence $[2, 2, 1, 0]$ is precisely the *Lehmer code* of 2310, appearing here as a byproduct of decomposition, without reference to ranking. It encodes the path through the factorial tree: at each level with m remaining digits, select the digit at index c_{m-1} (0-based), prepend it, and recurse.

9. Indexing

The permutation list subdivides recursively—first into n groups of $(n-1)!$, then each of those into $(n-1)$ groups of $(n-2)!$, and so on. Consequently *an index alone* encodes a complete path through this hierarchy.

The factorial tree structure allows us to map between a permutation's lexicographic index (0-based) and its Lehmer code directly. Each Lehmer code digit represents a left-to-right index at that level of the tree, guiding traversal down the factorial hierarchy. Given index i for $n = 4$ (0 to 23), the tree divides into 4 groups of $3! = 6$. The top-level digit is $k = \lfloor 17/6 \rfloor = 2$.

Example: Let $i = 17$ for $n = 4$. This index implicitly selects the third digit from $\{0, 1, 2, 3\}$ at the top level. Recurse with the remainder $i \bmod 6 = 5$ on the reduced set.

- $17 \div 6 = 2$ (remainder 5); select 2 (third in $\{0, 1, 2, 3\}$); remaining $\{0, 1, 3\}$.

- $5 \div 2 = 2$ (remainder 1); select 3 (third in $\{0, 1, 3\}$); remaining $\{0, 1\}$.
- $1 \div 1 = 1$ (remainder 0); select 1 (second in $\{0, 1\}$); remaining $\{0\}$.
- Select 0.

Yields 2310, with Lehmer code $[2, 2, 1, 0]$.

Thus the global lexicographic index can be decomposed using successive modulus operations $\text{mod}(n-1)!$, $\text{mod}(n-2)!, \ldots$. This is the factorial number system decomposition:

$$17 = 2 \cdot 3! + 2 \cdot 2! + 1 \cdot 1! + 0 \cdot 0!.$$

The coefficients form the Lehmer code, providing a bijection between indices and permutations.

This recursive extraction yields a sequence of positional choices, allowing recovery of the permutation at any given index i directly. In the next section, we convert this factorial tree structure into an explicit recursive algorithm.

10. From Structural Insight to a Recursive Permutation Algorithm

The recursive structure we've uncovered—grouping, digit removal, and relabeling—yields a generation procedure. We present it first in mathematical terms, then as code.

Mathematical Formulation

Let $A = (a_0, a_1, \ldots, a_{n-1})$ be a sorted sequence of n distinct elements. Let $\Pi(A)$ denote the sequence of all permutations of A in lexicographic order. Define $\Pi(A)$ recursively:

$$\Pi(A) = \begin{cases} \{A\} & \text{if } |A| = 1, \\ \bigcup_{i=0}^{n-1} \left(a_i \cdot \Pi(A \setminus \{a_i\})\right) & \text{otherwise,} \end{cases}$$

(where unions preserve lexicographic order by increasing i, ensuring lexicographic order), $A \setminus \{a_i\}$ is A with a_i removed (maintaining element order), and $a_i \cdot L$ prepends a_i to each permutation in L.

This mirrors the tree: branch on each possible leading digit, recurse on the remainder.

Example for $A = (0, 1, 2)$:

$$\Pi((0, 1, 2)) = 0 \cdot \Pi((1, 2)) \cup 1 \cdot \Pi((0, 2)) \cup 2 \cdot \Pi((0, 1)),$$

unfolding to the six permutations as before.

The recursion tree has $\sum_{k=0}^{n} n!/k!$ nodes, where each node represents a prefix of length k and the count reflects all partial and complete permutations generated during recursion.

Algorithmic Implementation

The following C++ code implements this recursion using a prefix buffer. It emits permutations at leaves and directly mirrors the structural decomposition. (Assume `MAX_N` defined, a global `int n` for the number of digits and a user-defined function `emit` to display each permutation.) Every line in the code corresponds directly to a step already encountered in our structural walkthrough.

```
void generate_permutations(const int* sorted_digits, int len,
    int* prefix, int depth, void (*emit)(int*, int)) {
    if (len == 1) {
        prefix[depth] = sorted_digits[0];   // base case: single digit left
        emit(prefix, n);                    // output one complete permutation
        return;
    }
    for (int i = 0; i < len; ++i) {         // iterate over groups of (len-1)!
        int first = sorted_digits[i];       // (a) select leading digit k
        prefix[depth] = first;              // record k in the output buffer

        int remaining[MAX_N];               // (b) remove k and build remaining[]
        int idx = 0;
        for (int j = 0; j < len; ++j)
            if (j != i) remaining[idx++] = sorted_digits[j];
        // (c) recurse on the (n-1)-set
        generate_permutations(remaining,len - 1, prefix, depth + 1, emit);
    }
}
```

This avoids swaps or backtracking by explicitly constructing reduced digit sets, preserving the structural clarity. Modifying the loop to select based on an index yields unranking (Section 9).

Comparison with Classical Methods

Classical recursive permutation generators (e.g., Sedgewick 1977) also fix prefixes and recurse, but typically use in-place swaps for efficiency. Our approach emphasizes structural recursion, aligning the code with the factorial tree and embedding properties. While less efficient than in-place methods, it provides clearer insight into permutation space organization, making it ideal for educational purposes.

11. The Compressed Codes

Compressed codes offer a compact representation of permutations, conceptually similar to factorial-based ranking schemes such as Lehmer codes and base conversions in combinatorial number systems. Each permutation can be compactly encoded by subtracting the numeric value of the lexicographically minimal permutation and dividing the result by $r-1$ (9 in base-10). These compressed codes preserve lexicographic order and allow exact reconstruction of the original permutations. (A full table of compressed codes for $n = 4$ is provided in Table 13.)

Definition

For any permutation p of n distinct digits written as an n-digit integer in radix r, we define its compressed code c as:

$$c = \frac{p - b_{\text{min}}}{r - 1}$$

where c is always an integer, b_{min} is the numeric value of the lexicographically minimal permutation of the digit set, and r is the numeral base (usually 10).

The formula is invertible:

$$p = (r-1)c + b_{\text{min}}$$

Since permutations are rearrangements of a fixed digit set, all share an identical digit sum S. Thus, by properties of positional number systems:

$$p \bmod (r-1) = S \bmod (r-1)$$

This guarantees that $p - b_{\text{min}}$ is divisible by $r-1$, making c always integral and fully invertible as shown.

Example:

Let $p = 1230$ and the digit set be $\{0, 1, 2, 3\}$. Then b_{min} = 0123, numerically interpreted as 123, and the compressed code is:

$$c = \frac{1230 - 123}{9} = \frac{1107}{9} = 123$$

To reconstruct the original permutation from the compressed code:

$$p = 9 \times 123 + 123 = 1107 + 123 = 1230$$

Ensure leading zeros are preserved during reconstruction to maintain fixed-width n-digit formatting.

Key Properties This compression scheme has several useful features:

- **Guaranteed divisibility**: The constant digit sum ensures divisibility of all permutation differences by $r-1$.
- **Lossless representation**: The mapping between permutations and compressed codes is bijective and fully reversible.
- **Structural inheritance**: *The list of compressed codes for n begins with the complete list for $n-1$*, preserving the recursive embedding of permutation structure across digit lengths.
- **Scalable boundary reconstruction**: The initial and final $k!$ permutations of a larger set S_n can be reconstructed directly from the compressed codes of S_k, interpreted in the permutation base of S_n, enabling efficient access to extremal regions.

Properties

From a practical standpoint, compressed codes scale well:

- Permutation values for $n = 10$ exceed 32-bit integer capacity, but the corresponding compressed codes remain within 32 bits.
- Even for $n = 19$, compressed codes fit within a 64-bit integer, whereas full permutation values do not.

Moreover, these codes establish a coordinate system that reflects the recursive organization of delta space.

12. Direct Compressed Code Generation

A variant of our prefix-based algorithm generates compressed codes without permutations, accumulating offsets using the same structural logic.

The key idea is to compute, at each recursive step, the numeric offset introduced by selecting a particular digit in the current position. Instead of assembling digits, this offset is computed and accumulate with the offsets from deeper recursive calls. (Assumes a global function `pow10` that returns 10^k for a given k.)

```
void generateCompressed(const int* digits, int len, int64_t* buffer, int& outIdx)
{
    if (len == 1) {
        buffer[outIdx++] = 0;
        return;
    }
    int sorted[MAX_N];
    memcpy(sorted, digits, len * sizeof(int));
    std::sort(sorted, sorted + len);

    int64_t base = 0;
    for (int i = 0; i < len; ++i)
        base = base * 10 + sorted[i];

    for (int i = 0; i < len; ++i) {
        int first = sorted[i];

        int remaining[MAX_N], idx = 0;
        for (int j = 0; j < len; ++j)
            if (j != i) remaining[idx++] = sorted[j];

        int64_t remainBase = 0;
        for (int j = 0; j < len - 1; ++j)
            remainBase = remainBase * 10 + remaining[j];

        // Offset: positional contribution of the leading digit and remaining tail
        int64_t offset = (int64_t(first) * pow10(len - 1) + remainBase - base) / 9;

        int start = outIdx;
        generateCompressed(remaining, len - 1, buffer, outIdx);
        for (int j = start; j < outIdx; ++j)
            buffer[j] += offset;
    }
}
```

Here, `digitsToInt` converts a list of digits to an integer, and `pow10(len - 1)` returns $10^{\text{len}-1}$. The division by 9 normalizes the positional contribution based on the change in base value from the sorted reference. As with the permutation generator, this algorithm makes $(n-1)!$ recursive calls per digit in the top-level list, traversing the same recursive tree but collecting offsets instead of digit sequences.

Each resulting integer in the `output` array is a compact representation of a permutation. (Note: For practical application, use tail recursion to avoid stack overflow. Additionally, preallocate working buffers to minimize memory churn, and consider avoiding per-call list copies by reusing or marking entries where feasible.)

Mathematical Formulation

We now present a formal description of the compressed code generator, assuming base-10 throughout.

Notation and Setup

Let $D = [d_0, d_1, \ldots, d_{n-1}]$ be a list of n distinct digits from the set $\{0, 1, \ldots, n-1\}$. Define:

- $\text{int}(D)$: the integer value of the digit list when interpreted as such, i.e.,

$$\text{int}([d_0, d_1, \ldots, d_{n-1}]) = \sum_{k=0}^{n-1} d_k \cdot 10^{n-1-k}$$

- $\text{base} = \text{int}(\text{sorted}(D))$: the lexicographically minimal permutation of D, interpreted as an integer.

Let $\text{remove}(D, d)$ denote the list obtained by removing the first occurrence of digit d from D, preserving the order of the remaining digits.

The compressed code function $\mathcal{K}(D)$ returns a list of nonnegative integers, one for each permutation of D in lexicographic order, defined recursively as follows:

$$\mathcal{K}(D) = \begin{cases} [0] & \text{if } |D| = 1, \\ \bigcup_{i=0}^{|D|-1} \{\text{offset}(d_i, D) + k \mid k \in \mathcal{K}(\text{remove}(D, d_i))\} & \text{otherwise,} \end{cases}$$

where:

$$\text{offset}(d_i, D) = \frac{d_i \cdot 10^{n-1} + \text{int}(\text{remove}(D, d_i)) - \text{base}}{9}$$

Each offset is added elementwise to the recursive result. The division by 9 yields exact integers due to the digit sum invariance of permutations, as explained in Section 11.

Example

Let $D = [0, 1, 2]$, so $r = 10$ and $\text{base} = \text{int}([0, 1, 2]) = 12$.

$$\begin{aligned} \text{offset}(0, [0,1,2]) &= \frac{0 \cdot 100 + \text{int}([1,2]) - 12}{9} = \frac{12 - 12}{9} = 0 \\ \text{offset}(1, [0,1,2]) &= \frac{1 \cdot 100 + \text{int}([0,2]) - 12}{9} = \frac{100 + 2 - 12}{9} = 10 \\ \text{offset}(2, [0,1,2]) &= \frac{2 \cdot 100 + \text{int}([0,1]) - 12}{9} = \frac{200 + 1 - 12}{9} = 21 \end{aligned}$$

Continuing recursively: $\mathcal{K}([0, 1, 2]) = [0, 1,\ 10, 12,\ 21, 22]$

13. Comparing Compressed Codes and Lexicographic Ranks

Table 2: Permutations of $n = 4$ with Lehmer and Compressed Codes

Index	Permutation	Lehmer Code	Base-10 Value	Compressed Code
0	0123	[0, 0, 0, 0]	123	0
1	0132	[0, 0, 1, 0]	132	1
2	0213	[0, 1, 0, 0]	213	10
3	0231	[0, 1, 1, 0]	231	12
4	0312	[0, 2, 0, 0]	312	21
5	0321	[0, 2, 1, 0]	321	22
6	1023	[1, 0, 0, 0]	1023	100
7	1032	[1, 0, 1, 0]	1032	101
8	1203	[1, 1, 0, 0]	1203	120
9	1230	[1, 1, 1, 0]	1230	123
10	1302	[1, 2, 0, 0]	1302	131
11	1320	[1, 2, 1, 0]	1320	133
12	2013	[2, 0, 0, 0]	2013	210
13	2031	[2, 0, 1, 0]	2031	212
14	2103	[2, 1, 0, 0]	2103	220
15	2130	[2, 1, 1, 0]	2130	223
16	2301	[2, 2, 0, 0]	2301	242
17	2310	[2, 2, 1, 0]	2310	243
18	3012	[3, 0, 0, 0]	3012	321
19	3021	[3, 0, 1, 0]	3021	322
20	3102	[3, 1, 0, 0]	3102	331
21	3120	[3, 1, 1, 0]	3120	333
22	3201	[3, 2, 0, 0]	3201	342
23	3210	[3, 2, 1, 0]	3210	343

Compressed codes and lexicographic ranks provide unique indices but from different principles: position vs. structure. They coexist in permutation space.

Classical rank and unrank methods treat permutations as positions within the lexicographic order, mapping each permutation to a unique lexicographic index via factorial decomposition (e.g., Lehmer code). This approach offers an exact, reversible mapping between permutations and their integer ranks, and is well suited for direct indexing, sampling, enumeration, and combinatorial testing.

Compressed codes, by contrast, encode each permutation's absolute offset from the lexicographic minimum, using the simple arithmetic formula introduced earlier. Unlike rank, this representation is structurally grounded and works uniformly across numeral systems, varying digit ranges, and digit set offsets (e.g., non-zero-starting sets).

Taken together, these representations offer complementary strengths: lexicographic rank provides precise positional access, while compressed codes support recursive composability and structural insight. Rank and unrank remain unmatched in their utility. Compressed codes are useful for theoretical exploration and, as we will see in the next section, can mark structural transitions.

14. Structural Insights and Logarithmic Behavior of Compressed Codes

Building on the foundational properties of compressed codes, inspection reveals patterns that align with permutation space. Just as we divided the permutations of S_n into n groups of $(n-1)!$ permutations each in *Factorial Partitioning of Permutation Space* (7), they can just as naturally be grouped by successive factorial blocks. As we now show, these codes in base-10 align precisely with the start of these blocks.

Factorial Boundary Alignments and Anchor Points

Inspection reveals that base-10 codes of the form 10^k (for $k = 0, 1, 2, \ldots$) consistently mark the starting positions of factorial-sized blocks within the compressed code list when enumerating permutations of S_n. For example, in S_n for $n \geq 6$, the codes 1, 10, 100, 1000, and 10000 align with the first permutations of the 2!, 3!, 4!, 5!, and 6! blocks respectively.

More significantly, for $k \geq 4$, these anchor codes mark structural transitions, and decode to permutations whose *leading digit shifts to 1*—signaling a change in the recursive embedding of permutation space. This pattern appears stable across values of n and may continue indefinitely. (Requires further verification).

This phenomenon further reinforces the recursive logic underlying compressed codes and aligns with the trailing-digit regularities discussed in *Observed Structural Regularities at Factorial Boundaries* (Appendix F). It reflects the hierarchical decomposition of permutation space and highlights how compressed codes embed a coarse-grained structural map within their digit patterns. Viewed this way, the codes function as both identifiers and signposts within a stratified geometric landscape.

Logarithmic Scaling Analogy: Compressed codes scale additively in length, evoking how logarithms compress exponential growth. Although digit count grows as $O(\log_r(n!)) \sim O(n \log n / \log r)$, it remains compact relative to the factorial growth of permutation space.

Example: For $n = 4$ (24 permutations), codes range from 0 to 343 (up to 3 digits); for $n = 5$ (120 permutations), up to 4 digits suffice—capturing the $5 \times 4!$ growth with minimal increase in code length.

Implications and Applications: Changes in codes' digit count provide predictable waypoints in the recursive structure of permutation space, enabling prefix-based navigation, structural analysis, and efficient storage. This supports natural bucketing of permutations and facilitates recursive algorithms that span multiple sizes. For instance, codes from S_n extend into S_{n+1} when decoded using the base of S_{n+1}, maintaining structural continuity across layers.

This perspective supports exploratory tools such as similarity metrics, probabilistic sampling, and generalizations to multisets. Collectively, these capabilities underscore the role of compressed codes in enumeration and structural analysis.

15. Recursive Symmetries in Compressed Code Delta Space

Building on the symmetries in permutation delta space (Section 2), we extend the analysis to deltas between compressed codes. Define the compressed delta as the numeric difference between successive compressed codes C in lexicographic order:

$$\Delta_i^c = C_{i+1} - C_i,$$

yielding a list of length $n! - 1$. These deltas exhibit the same recursive and mirror symmetries as permutation deltas, but scaled by a factor of $1/(r-1)$ (where r is the base, typically 10).

Delta Table for Compressed Codes of 4-Digit Permutations:

Index	Delta Expression	
0:	$1 - 0 = 1$	
1:	$10 - 1 = 9$	
2:	$12 - 10 = 2$	
3:	$21 - 12 = 9$	
4:	$22 - 21 = 1$	
5:	$100 - 22 = 78$	
6:	$101 - 100 = 1$	
7:	$120 - 101 = 19$	
8:	$123 - 120 = 3$	
9:	$131 - 123 = 8$	
10:	$133 - 131 = 2$	
11:	$210 - 133 = 77$	<-- centre of symmetry (pivot)
12:	$212 - 210 = 2$	
13:	$220 - 212 = 8$	
14:	$223 - 220 = 3$	
15:	$242 - 223 = 19$	
16:	$243 - 242 = 1$	
17:	$321 - 243 = 78$	
18:	$322 - 321 = 1$	
19:	$331 - 322 = 9$	
20:	$333 - 331 = 2$	
21:	$342 - 333 = 9$	
22:	$343 - 342 = 1$	

The sequence partitions naturally: the first five match the deltas for $n = 3$, the central 13 are unique (centered on index 11), and the final five mirror the first. As it turns out, compressed code deltas are the permutation deltas from Table 2 scaled by $\frac{1}{9}$, providing an alternative path to code generation via accumulation of the reduced deltas.

We observe the same two intertwined symmetries in compressed code delta space that we saw in permutation delta space, the structural recursion and arithmetic reflection around a central pivot.

The mirror symmetry satisfies:

$$\Delta^c_{\text{pivot}+d} = \Delta^c_{\text{pivot}-d}$$

for valid d, with the pivot at index $\frac{n!-2}{2}$.

Recursive construction mirrors that of Δ^p_n as discussed in Section 4.

$$\Delta^c_n = \Delta^c_{n-1} \parallel C^c_n \parallel \Delta^c_{n-1},$$

where C^c_n is the scaled central spline (e.g., $C^c_4 = [78, 1, 19, 3, 8, 2, 77, 2, 8, 3, 19, 1, 78]$ for $n = 4$).

Implications

Reflection in code space simplifies to:

$$C_{\text{ref}} = C_{\text{max}} - C,$$

where C_{max} is the compressed code for the last permutation in the set, enabling negligible-cost pairing.

These deltas support direct generation algorithms, delta-encoded compression, and sampling at factorial boundaries. They also suggest combinatorial links, such as interpreting small deltas as minimal changes in a reduced factorial system.

16. Positioning Within the Broader Combinatorics Landscape

We position our framework within the broader field of combinatorics. Classical approaches often treat permutations as flat, linear structures; our approach reveals them as recursively nested. This aligns with Gray codes and finite differences, but our delta symmetry encodes deeper, more intricate patterns.

This compares to with algebraic families like generating trees, Catalan structures, and Young tableaux, which also exhibit self-similar growth. Our framework is arithmetic but suggests connections.

17. Conclusion and Future Work

This work began as a search for a compressed code algorithm, as conjectured in Taylor (2011), and culminated in its discovery. More significantly, it revealed a structurally grounded method for permutation generation—distinct from classical approaches—arising from the compositional geometry of lexicographic order.

We also formalized the recursive and symmetric structure of permutation delta space, including midpoint mirroring, prefix nesting, and factorial block decomposition. This framework enables mirrored permutations to be computed with negligible cost, effectively doubling output without additional computation.

Compressed codes naturally emerge within this structure as compact, reversible representations. They reflect the recursive logic of permutation growth and offer an efficient coordinate system aligned with permutation space.

The core contribution is conceptual: a shift from procedural generation to structural insight. The generation algorithm is a consequence of this perspective, rather than its primary objective. Together, these ideas suggest several paths for further exploration:

- **Generalization Beyond Base-10 Digits**: Extend the method to arbitrary alphabets by adapting the factorial boundaries, delta rules, and encoding scheme—enabling compression of non-numeric permutations.
- **Direct Random Access**: Investigate whether the recursion underlying compressed codes can improve ranking/unranking or enable more direct access to targeted segments of permutation space.
- **Cryptographic Applications**: Explore the use of recursively compressed permutation spaces in cryptographic functions, particularly for pseudorandom generation and keyspace structuring.
- **Compressed Storage and Indexing**: Combine compressed codes with other combinatorial representations to enable efficient storage, retrieval, and filtering of large permutation sets.
- **Sampling and Navigation**: Develop probabilistic and indexed traversal methods within compressed code space, supporting partial decoding or local permutation access.
- **Factorial Digit Regularities**: Analyze the factorial-aligned digit regularities observed during empirical exploration, and determine whether deeper arithmetic structure governs global behavior in permutation space.

These avenues underscore the value of structural recognition in permutation space. While recursion and mirroring are latent in lexicographic order, their implications for compression, indexing, and symbolic manipulation remain largely untapped. This framework bridges combinatorial theory and algorithmic design, offering a foundation for continued structural exploration.

References

1. Stanley, R. P. *Enumerative Combinatorics, Volume 1.* Cambridge University Press, 2012.
   A foundational text covering enumeration techniques, bijective proofs, and recursive methods in combinatorics.

2. Knuth, D. E. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1.* Addison-Wesley, 2011.
   Canonical reference for permutation generation, including lexicographic methods and ranking/unranking algorithms.

3. Sedgewick, R. "Permutation Generation Methods." *ACM Computing Surveys*, vol. 9, no. 2, 1977, pp. 137–164.
   A comprehensive survey of classical permutation algorithms.

4. Heap, B. R. "Permutations by Interchanges." *The Computer Journal*, vol. 6, no. 3, 1963, pp. 293–298.
   Original presentation of Heap's permutation generation algorithm.

5. Johnson, S. M. "Generation of Permutations by Adjacent Transposition." *Mathematics of Computation*, vol. 17, no. 83, 1963, pp. 282–285.
   Source of the Johnson–Trotter algorithm, also known as Steinhaus–Johnson–Trotter.

6. Lehmer, D. H. "Teaching Combinatorial Tricks to a Computer." *Proceedings of the Symposia in Applied Mathematics*, vol. 10, 1960, pp. 179–193.
   A key reference for Lehmer code and its role in ranking/unranking permutations.

7. Savage, C. D. "A Survey of Combinatorial Gray Codes." *SIAM Review*, vol. 39, no. 4, 1997, pp. 605–629.
   A comprehensive overview of Gray codes and their combinatorial applications.

8. Jordan, C. *Calculus of Finite Differences.* Chelsea Publishing Company, 1965.
   Classic reference on finite difference methods, foundational for analyzing numerical variation in sequences.

9. Taylor, G. W. "A Permutation on Combinatorial Algorithms." 2011.
   https://tropicalcoder.com/APermutationOnCombinatorialAlgorithms.htm
   Original formulation of the recursive self-similarity structure in permutation sequences.

10. Felsner, S., & Valtr, P. "Coding and Counting Arrangements of Pseudolines." *Discrete & Computational Geometry*, vol. 46, no. 3, 2011, pp. 405–416.
    For generating trees and structural encodings in recursive families.

11. Stanley, R. P. *Catalan Numbers.* Cambridge University Press, 2015.
    Authoritative, modern reference on Catalan structures.

12. Fulton, W. *Young Tableaux: With Applications to Representation Theory and Geometry.* Cambridge University Press, 1997.
    For recursive tableaux structures and their combinatorial significance.

Appendix A: Python Permutation Generator

This Python code implements the prefix-based permutation generator. It directly emits lexicographic permutations and can serve as a reference for experimentation or adaptation. (Code listing 1. follows below. A digital copy is also available at:
https://tropicalcoder.com/PermutationGenerator.py)

Appendix B: Basic Python Compressor and Reconstructor

Minimal Python code for compressed code generation and permutation reconstruction. Compressed code generation typically operates as a buffered recursive process. (Code listing 2. follows below, but a digital version is available at:
https://tropicalcoder.com/CompressedCodeGenerator.py) PermutationGenerator.py

Appendix C: C++ Permutation Generators with Index Column

The following C++ programs generate full three-column permutation tables. Each outputs: the permutation index, the generated permutation (via direct construction), and a validation permutation generated using `std::next_permutation`, allowing complete verification of algorithm correctness.

This version generates the full sequence of lexicographic permutations from index zero upward. A digital copy is available at:
https://tropicalcoder.com/PermutationGenerator.cpp

This variation accepts an initial index and generates permutations starting from that point. This enables targeted exploration of specific regions within the permutation space, useful for analyses such as midpoint studies or recursive sampling. A digital copy is available at:
https://tropicalcoder.com/PermutationGeneratorIndexed.cpp

Appendix D: C++ Full Three-Column Generator

The following C++ program demonstrates compressed code generation, permutation reconstruction, and validation. The output displays three columns: Compressed Code, Reconstructed Value, and Actual Permutation (with leading zeros). A digital copy is available at:
https://tropicalcoder.com/fractal_permutation_generator.cpp

Appendix E: Permutation Table Reuse Demo

The following C++ program generates compression codes from small-n sets and applies them to reconstruct both first and last small-$n!$ permutations of larger-n sets. A digital copy is available at:
https://tropicalcoder.com/FractalPermReconstructionDemo.cpp

The following Python program performs the same operation in Python: A digital copy is available at:
https://tropicalcoder.com/reverseCodes.py

Appendix F: Observed Structural Regularities at Factorial Boundaries

This appendix reports recurring numerical patterns observed during real-time enumeration of permutations—a phenomenon not yet fully characterized, but one that suggests deeper combinatorial structure and invites further theoretical investigation.

Empirical Observation

In one experiment, we enumerated permutations of $n = 20$ in real time. To slow the output to a readable speed, the program was configured to print every $10!$th permutation. Surprisingly, many of the displayed permutations ended in the digit 19—deviating from the anticipated rapid variation. The same trailing-digit stability appeared when sampling every $8!$th, $9!$th, or $11!$th permutation, suggesting the key lies in choosing sampling of comparable factorial order. A representative sample is shown below, with each index followed by its compressed code (in base-21, using digits 0—9 and letters A—K) and corresponding permutation.

Compressed Codes and Permutations Sampled at $10!$ Intervals (from $n = 20$)

Index	Code	Permutation																			
0	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
3628800	1000000000	0	1	2	3	4	5	6	7	8	10	9	11	12	13	14	15	16	17	18	19
7257600	2100000000	0	1	2	3	4	5	6	7	8	11	9	10	12	13	14	15	16	17	18	19
10886400	3210000000	0	1	2	3	4	5	6	7	8	12	9	10	11	13	14	15	16	17	18	19
14515200	4321000000	0	1	2	3	4	5	6	7	8	13	9	10	11	12	14	15	16	17	18	19
18144000	5432100000	0	1	2	3	4	5	6	7	8	14	9	10	11	12	13	15	16	17	18	19
21772800	6543210000	0	1	2	3	4	5	6	7	8	15	9	10	11	12	13	14	16	17	18	19
25401600	7654321000	0	1	2	3	4	5	6	7	8	16	9	10	11	12	13	14	15	17	18	19
29030400	8765432100	0	1	2	3	4	5	6	7	8	17	9	10	11	12	13	14	15	16	18	19
32659200	9876543210	0	1	2	3	4	5	6	7	8	18	9	10	11	12	13	14	15	16	17	19
36288000	A987654321	0	1	2	3	4	5	6	7	8	19	9	10	11	12	13	14	15	16	17	18
39916800	10000000000	0	1	2	3	4	5	6	7	9	8	10	11	12	13	14	15	16	17	18	19
43545600	12000000000	0	1	2	3	4	5	6	7	9	10	8	11	12	13	14	15	16	17	18	19
47174400	13100000000	0	1	2	3	4	5	6	7	9	11	8	10	12	13	14	15	16	17	18	19
50803200	14210000000	0	1	2	3	4	5	6	7	9	12	8	10	11	13	14	15	16	17	18	19
54432000	15321000000	0	1	2	3	4	5	6	7	9	13	8	10	11	12	14	15	16	17	18	19
58060800	16432100000	0	1	2	3	4	5	6	7	9	14	8	10	11	12	13	15	16	17	18	19
61689600	17543210000	0	1	2	3	4	5	6	7	9	15	8	10	11	12	13	14	16	17	18	19
65318400	18654321000	0	1	2	3	4	5	6	7	9	16	8	10	11	12	13	14	15	17	18	19
68947200	19765432100	0	1	2	3	4	5	6	7	9	17	8	10	11	12	13	14	15	16	18	19
72576000	1A876543210	0	1	2	3	4	5	6	7	9	18	8	10	11	12	13	14	15	16	17	19
76204800	1B987654321	0	1	2	3	4	5	6	7	9	19	8	10	11	12	13	14	15	16	17	18

This result was initially puzzling. Since the last digit varies most rapidly in lexicographic order, a more even distribution was expected. In hindsight, the observed *stability zones*—where the trailing digit remains fixed—appear to reflect an underlying structural cause. These zones seem to arise from the hierarchical structure of the factorial number system, which governs the evolution of lexicographically ordered permutations. The recurring patterns at factorial strides suggest a form of long-range order in permutation space.

Interpretation in Permutation Space

We suggest these observations can be understood directly in terms of the grouping structure of permutation space. In the factorial number system (FNS), each permutation's lexicographic rank corresponds to a factorial-based expansion. Sampling every $m!$ steps for $m \ll n$ is equivalent to varying the $n - m$ most significant factorial digits, while the least significant digits remain temporarily stable until carries occur.

This mirrors how coarse strides in base-10 fix trailing digits in decimal numbers. The reoccurrence of trailing digits across many samples arises from the hierarchical traversal inherent in the factorial number system. Viewed through this lens, what initially appeared as surprising numerical behavior becomes a predictable artifact of recursive enumeration.

The observed reoccurrence of trailing digits is not an isolated anomaly, but reflects deeper structural regularities in permutation space. Lexicographic order maps directly to numeric value in the FNS, making such repetition an inherent outcome of its recursive encoding. Each stride of size $10!$ for $n = 20$ lands in a new factorial block, where the lower-order digits of the permutation (the suffix) tend to remain unchanged over consecutive samples. This arises because the lex-order tree is recursively partitioned: $n!$ permutations are divided into n blocks of $(n-1)!$, then into $(n-2)!$, and so on. Sampling at factorial intervals traverses this hierarchy at coarse resolution, creating localized regularities in the trailing digits—both in the permutations and their compressed representations.

Hierarchical Stability in Compressed Codes

Compressed codes reflect lexicographic rank directly. When expressed in a higher base such as 21, they reveal a denser lattice of anchor points, enhancing the visibility of structural transitions aligned with factorial boundaries. (Base-21 allows division by $r-1 = 20$, yielding an integer aligned with rank.)

When the same permutation samples are examined through their compressed codes in base-21, a second pattern emerges. The codes display numerical anchors of the form `100...0`, `210...0`, `3210...0`, and so on—each aligned precisely with a factorial boundary. These values correspond to base-21 prefixes multiplied by descending powers of the base, e.g., $1 \cdot 21^{11}$, $(2 \cdot 21 + 1) \cdot 21^{10}$, $(3 \cdot 21^2 + 2 \cdot 21 + 1) \cdot 21^9$, etc. The samples were taken at multiples of $10!$ starting from $132 \times 10!$, a point where the digit width of the codes has expanded.

Index	Compressed Code (base-21)	Expression as Power of 21
479001600	`100000000000`	$1 \cdot 21^{11}$
958003200	`210000000000`	$43 \cdot 21^{10}$
1437004800	`321000000000`	$1366 \cdot 21^9$
1916006400	`432100000000`	$35903 \cdot 21^8$
2395008000	`543210000000`	$752388 \cdot 21^7$
2874009600	`654321000000`	$12038259 \cdot 21^6$
3353011200	`765432100000`	$144459084 \cdot 21^5$
3832012800	`876543210000`	$1284995385 \cdot 21^4$
4311014400	`987654321000`	$8355849446 \cdot 21^3$
4790016000	`A98765432100`	$39754765487 \cdot 21^2$
5269017600	`BA9876543210`	$132515173979 \cdot 21$
5748019200	`CBA987654321`	N/A
6227020800	`1000000000000`	$1 \cdot 21^{12}$

Conjecture: Stability of Suffixes at Factorial Intervals

Suffix stability emerges when sampling permutation space at factorial intervals, reflecting the hierarchical structure of lexicographic ordering.

For large n, such sampling induces coherence in the final digits of both the permutation and its compressed code. We posit that this stability might enable algorithmic advantages in prediction, sampling, and structure-aware hashing, by exploiting coarse-grained regularities across large permutation sets.

Perspective and Future Work.

The observed regularity appears to reflect a combinatorial invariant rooted in recursive structure. The coherence across factorial intervals likely stems from the internal nesting of permutation cosets, where each level progressively fixes more digits and restricts local variation.

Quantitative analysis could clarify this effect—for instance, by measuring digit variance across fixed factorial strides—to assess the degree and persistence of suffix stability. Such work may reveal how hierarchical traversal interacts with structural regularities in permutation space.

Further investigation could explore whether analogous stability patterns arise under alternate orderings (e.g., colex), or in nonuniform encodings. These phenomena may have algorithmic applications in structure-aware sampling, predictive indexing, or compression, and could invite formal treatment via algebraic combinatorics or spectral analysis of permutation graphs.

Appendix G: Recursive Call Structure

The algorithm introduced here explores a recursive tree of choices, making exactly $n!$ terminal calls—one at each leaf node—implicitly computing the factorial through structural descent.

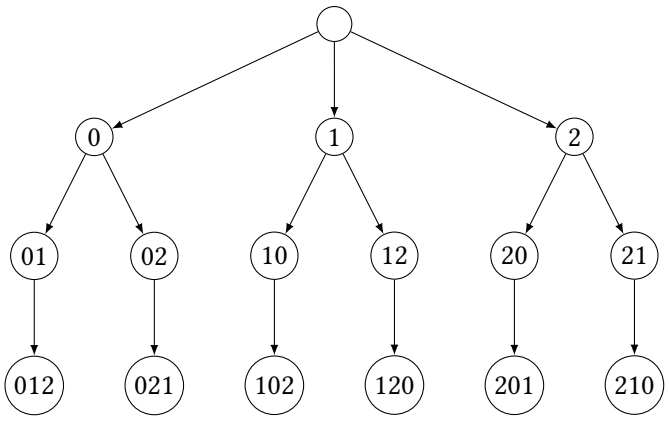

Figure 1: Recursive permutation tree for $\{0, 1, 2\}$. Each path from root to leaf corresponds to one permutation.

Listing 1: Permutation Generator

```
import itertools
MAX_N = 20
n = 20  # <-- Set n here
PRINT_INTERVAL = 1000000  # <-- Print every million permutations
global_index = 0

reference_perms = itertools.permutations(range(n))

def print_perm(perm, end=''):
    for num in perm:
        print(f"{num:2d} ", end='')
    print(end, end='')

def emit(prefix):
    global global_index

    ref = next(reference_perms)
    global_index += 1

    if global_index % PRINT_INTERVAL == 0:
        print(f"{global_index:12d}  ", end='')
        print_perm(prefix, end='  ')
        print_perm(ref)
        print()

# Recursive permutation generator
def generate_permutations(sorted_digits, prefix):
    if not sorted_digits:
        emit(prefix)
        return

    for i, first in enumerate(sorted_digits):
        remaining = sorted_digits[:i] + sorted_digits[i+1:]
        generate_permutations(remaining, prefix + [first])

# Entry point
def main():
    digits = list(range(n))
    digits.sort()  # Initial sort
    generate_permutations(digits, [])

if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        print("Any key exits.")
```

Listing 2: Compressed Code Generator

```
def digits_to_int(digits):
    result = 0
    for d in digits:
        result = result * 10 + d
    return result

def generate_compressed(digits):
    if len(digits) == 1:
        return [0]

    sorted_digits = sorted(digits)
    base_value = digits_to_int(sorted_digits)
    result = []

    for i, first in enumerate(sorted_digits):
        remaining = sorted_digits[:i] + sorted_digits[i+1:]
        remain_base = digits_to_int(remaining)
        power_term = 10 ** (len(digits) - 1)
        offset = (first * power_term + remain_base - base_value) // 9

        sub_result = generate_compressed(remaining)
        for val in sub_result:
            result.append(offset + val)

    return result

def reconstruct_permutation(code, base):
    return code * 9 + base

if __name__ == "__main__":
    digits = [0, 1, 2, 3, 4, 5]
    base = digits_to_int(digits)
    compressed = generate_compressed(digits)

    for code in compressed:
        reconstructed = reconstruct_permutation(code, base)
       print(f"Code: {code}, Reconstructed: {str(reconstructed).zfill(
          len(digits))}")
```