arXiv:2505.09628v2 [cs.DM] 16 May 2025

An $\mathcal{O}(n)$ Space Construction of Superpermutations

Dhruv Ajmera[1]

[1] Lone Star High School, Panther Creek Pkwy, Frisco, 75033, Texas, United States.

Contributing authors: dhruvajmera0@gmail.com;

Abstract

A superpermutation is a sequence that contains every permutation of $\boldsymbol{n}$ distinct symbols as a contiguous substring. For instance, a valid example for three symbols is a sequence that contains all six permutations. This paper introduces a new algorithm that constructs such sequences more efficiently than existing recursive and graph-theoretic methods. Unlike traditional techniques that suffer from scalability and factorial memory demands, the proposed approach builds superpermutations directly and compactly. This improves memory usage, enabling the construction of larger sequences previously considered impractical.

Keywords: permutation, superpermutation, pseudo-random, algorithm

1 Introduction

The combinatorial structure of superpermutations—strings that embed all $n!$ permutations of a symbol set as contiguous substrings—has fascinated researchers due to its connections with encoding theory, sequence compression, and Hamiltonian path constructions. Traditional investigations focus on minimizing the length of such sequences, a goal that has produced both theoretical bounds and computational challenges.

This paper investigates a less-explored but practically important aspect of these sequences: optimizing the algorithm utilized to generate them. Existing methods for constructing superpermutations—typically recursive or graph-theoretic—require cumulative construction of subcases [1, 2] or the discovery of shortest Hamiltonian paths [1, 3, 4]. These strategies suffer from factorial time complexity coupled with factorial (or worse) space complexity, making them infeasible at even minor scales. In contrast, our method constructs superpermutations through a deterministic chaining of beads. It doesn't require the construction of subcases and incrementally outputs

the result, avoiding storing the entire sequence in memory because of pre-calculated overlap lengths (a feat that is not feasible with recursive methodologies). The algorithm achieves a worst-case runtime of $\mathcal{O}(n!)$ with only $\mathcal{O}(n)$ space usage. Details of the implementation, theoretical properties, and comparative benchmarks follow in subsequent sections.

2 Results

We analyze the performance of our algorithm through asymptotic comparison with both recursive and graph-theoretic approaches to constructing superpermutations. Theoretical runtime and space complexity expressions for each method are derived in Section 6. While recursive methods operate in $\mathcal{O}(n!)$ time, graph-theoretic approaches are significantly worse, exhibiting time complexity on the order of $\mathcal{O}(2^{n!}n!^2)$ due to the combinatorial explosion in possible paths through the graph of permutations. Both methods also suffer from high space complexity: $\mathcal{O}(n!)$ for recursive methods and $\mathcal{O}(n!^2)$ for graph-theoretic techniques. In contrast, our algorithm's incremental construction allows it to run in $\mathcal{O}(n!)$ time using only $\mathcal{O}(n)$ space, as it maintains only the current working bead and a minimal buffer.

To contextualize these differences, Table 1 summarizes the high-level asymptotic behavior of each approach. A more detailed breakdown and derivation of these time and space complexities is outlined in Section 6.

Table 1 Asymptotic time and space complexity of three construction methods.

Method	**Time Complexity**	**Space Complexity**
This Work	$\mathcal{O}(n!)$	$\mathcal{O}(n)$
Recursive	$\mathcal{O}(n!)$	$\mathcal{O}(n!)$
Graph-Theoretic	$\mathcal{O}(2^{n!}n!^2)$	$\mathcal{O}(n!^2)$

3 Framework

3.1 Beads

Firstly, we fix an integer $n \geq 3$. Then, our alphabet is the ordered tuple $s = (c_1, c_2, \ldots, c_n)$, with each c_k being a *letter* in the alphabet.

Next, let $\langle \cdot \rangle : s^* \longmapsto \Sigma^*$ be the concatenation map that sends a finite tuple $(x_1, \ldots, x_k)$ of letters to the string $x_1 x_2 \ldots x_k$, where s^* is the set of all finite tuples over s and Σ^* is the set of strings.

Henceforth, we identify a tuple with its flattened string without further comment. Additionally, we identify tuple entries x_i with their corresponding letters in the flattened string.

Now, we begin by introducing the core building block of our construction: the *bead*. Beads serve as compact containers for multiple permutations and are designed

to maximize space efficiency while maintaining structural simplicity. This allows us to build superpermutations with predictable overlap behavior.

Definition 1 (Bead) A *bead* is a $(2n-1)$-tuple

$$b = (x_1, \ldots, x_{2n-1})$$

such that every length-n window $(x_j, \ldots, x_{j+n-1})$, $1 \leq j \leq n$,

1. contains each letter $c_1, \ldots, c_n$ exactly once (i.e. is a permutation), and
2. the n windows are pairwise distinct.

We denote the set of all such tuples as $\mathcal{R}^0$.

Now, we introduce a formal notion of what space efficiency in encoding permutations is and how to quantify it.

Definition 2 (Space Efficiency) Space efficiency is the ratio of the length of a string to the number of permutations it contains. Lower ratios indicate greater space efficiency.

We now establish the fundamental properties of beads. These results demonstrate that beads are optimally compact and that they exhibit predictable behavior, properties that we will later exploit when constructing larger structures. Part (i) of this proof motivates our usage of beads as the base structure for construction.

Theorem 1 (Bead Properties) The following properties hold for all beads.

(i) A bead is the most space-efficient structure for containing permutations.
(ii) Exactly $(n-1)!$ distinct beads are needed in a superpermutation for n.
(iii) Every mth letter of a bead, such that $n < m \leq 2n-1$, is a copy of the $m-n$th letter.

Proof We prove each statement individually.

(i) *A bead is the most space-efficient structure for containing permutations.*
Consider b_1 of length $2n-1$. Appending x_k (where $k \leq n$) duplicates an existing permutation, so no additional permutations can be added in this manner.

Furthermore, a string of length ℓ has at most $\ell - n + 1$ distinct length-n windows (the window can begin only at positions $1, \ldots, \ell-n+1$), so $\ell-n+1 \geq n$ and therefore $\ell \geq 2n-1$. Because a bead has a length of exactly $2n-1$, no shorter string can realize the required windows, making the bead length-minimal and thus space-optimal.

(ii) *We require $(n-1)!$ distinct beads in a superpermutation for n.*
Each bead contains n distinct permutations. Since we require $n!$ distinct permutations, the number of beads required is $\frac{n!}{n} = (n-1)!$.

(iii) *Every mth letter of a bead, such that $n < m \leq 2n-1$, is a copy of the $(m-n)$th letter.*
By Definition 1 the windows $x_1 \ldots x_n$ and $x_{m-n+1} \ldots x_m$ must differ. This implies that for any index $m > n$, $x_m = x_{m-n}$. □

From part (iii) of the previous proof, we will now notate any bead b as

$$b = (x_1, \ldots, x_n, x_1, \ldots, x_{n-1}),$$

due to the symmetry of letters about either side of the n-th letter.

Thus, we use beads as our base case due to their inherent compactness in construction.

3.2 Intersections

To efficiently construct superpermutations, we must transition from one bead to the next while utilizing as much of the previous structure as possible. This is achieved via controlled overlaps between beads, which we formalize as intersections.

Definition 3 (Intersection) An intersection of length ℓ is a common substring that is shared between two structures.

The following definition will be for ease of reference throughout future proofs.

Definition 4 (Leading letter) The leading letter of some sequence is the letter that occupies the first index of that sequence.

Now, we define a function that enables us to transition between beads, as will be proven later. Thus, we are able to establish the first steps of generating the superpermutation from successive beads.

Definition 5 (Straight-shift SS_{n-k}) Let $1 \leq k \leq n-2$ and $b = (x_1, \ldots, x_{2n-1}) \in \mathcal{R}^0$. Define $y = \mathrm{SS}_{n-k}(b)$ as:

$$y_m = \begin{cases} x_{m+1}, & 1 \leq m < n-k, \\ x_1, & m = n-k, \\ x_m, & n-k < m \leq n, \\ y_{m-n}, & n < m \leq 2n-1. \end{cases}$$

An example of the straight-shift function at index $n-2$ is shown below for the set $\{c_1, c_2, \ldots, c_5\}$.

$$c_1c_2c_3c_4c_5c_1c_2c_3c_4 \longrightarrow c_2c_3c_1c_4c_5c_2c_3c_1c_4$$

Fig. 1 Application of straight-shift SS_{n-2} on a bead over $\{c_1, c_2, \ldots, c_5\}$, where the leading letter is re-inserted and structural duplication is applied.

Definition 6 (Straight-unshift SU_{n-k}) Let $1 \leq k \leq n-2$ and $b = (x_1, \ldots, x_{2n-1}) \in \mathcal{R}^0$. Define $y = \mathrm{SU}_{n-k}(b)$ as:

$$y_m = \begin{cases} x_{n-k}, & m = 1, \\ x_{m-1}, & 1 < m \leq n-k, \\ x_m, & n-k < m \leq n, \\ y_{m-n}, & n < m \leq 2n-1. \end{cases}$$

An example of the straight-unshift function at index $n-2$ is shown below for the set $\{c_1, c_2, \ldots, c_5\}$.

$$c_1c_2c_3c_4c_5c_1c_2c_3c_4 \longrightarrow c_3c_1c_2c_4c_5c_3c_1c_2c_4$$

Fig. 2 Application of straight-unshift SU_{n-2} on a bead over $\{c_1, c_2, \ldots, c_5\}$, where the $n-2$th letter is re-inserted and structural duplication is applied.

Lemma 1 (SS_{n-k} And SU_{n-k} Are Inverses) Let $1 \leq k \leq n-2$. Then,

$$\mathrm{SS}_{n-k} = (\mathrm{SU}_{n-k})^{-1}$$

and

$$\mathrm{SU}_{n-k} = (\mathrm{SS}_{n-k})^{-1}$$

hold true.

Proof Write $b = (x_1, \ldots, x_n, \ldots, x_{n-1})$. Applying SS_{n-k}, we obtain

$$\mathrm{SS}_{n-k}(b) = (x_2, x_3, \ldots, x_1, x_{n-k}, \ldots, x_n, \ldots, x_{n-1}).$$

Applying SU_{n-k} onto this bead, we obtain

$$\mathrm{SU}_{n-k}(\mathrm{SS}_{n-k}(b)) = (x_1, \ldots, x_n, \ldots, x_{n-1}),$$

proving the inverse in one direction. Similarly, apply SU_{n-k} to b first, obtaining

$$\mathrm{SU}_{n-k}(b) = (x_{n-k}, x_1, x_2, \ldots, x_{n-k-1}, x_{n-k+1}, \ldots, x_n, \ldots, x_{n-1}).$$

Finally, by applying SS_{n-k} onto this bead, we obtain

$$\mathrm{SS}_{n-k}(\mathrm{SU}_{n-k}(b)) = (x_1, \ldots, x_n, \ldots, x_{n-1}),$$

proving the inverse in both directions. Hence, SS_{n-k} and SU_{n-k} are mutual inverses. □

The following notation will be introduced for ease throughout the paper.

Notation 1 (Function Exponentiation) For any function f on a bead b at index i,

$$f_i^j(b) := \underbrace{f_i\Big(f_i\big(\cdots(f_i(b))\big)\Big)}_{j \text{ times}}.$$

Note that this is the notation used in Axler [5]; we adopt it to avoid clunky superscripts such as $\mathrm{SS}_{n-k}^{(j)}$.

The following theorem formally relates our previously defined straight-shift function to the process of transitioning between different beads with a fixed intersection length corresponding to a straight-shift at a given index. In addition, it provides a preliminary basis for calculating the final length of the generated sequence via the number of total intersections.

Theorem 2 (Intersection Properties Theorem) The following properties hold for all intersections:

(i) The maximum intersection length between two beads is $\ell = n - 2$.
(ii) We require $(n-1)! - 1$ intersections in a superpermutation for n.

Proof Consider $b_1 = (x_1, \ldots, x_n, x_1, \ldots, x_{n-1})$.

(i) *The maximum intersection length between two beads is $\ell = n - 2$.*
We will first prove that any intersection length such that $n \geq \ell > n - 2$ is impossible by contradiction, and then directly prove that $\ell = n - 2$ is possible.

Using the last n letters of b_1 as the overlap with b_2 can be ruled out, since this would result in a repeated permutation, contradicting Definition 1.

Now, assume an overlap of length $n - 1$. Then the suffix $x_2 \ldots x_n$ of b_1 would equal the prefix $x_1 \ldots x_{n-1}$ of b_2, forcing $x_1 = x_2$. This contradicts Definition 1, so an overlap of length $n - 1$ is impossible.

However, the last $n - 2$ letters of b_1 can be used as the start of the first permutation of b_2. To do so, we duplicate b_1 and straight-shift at the $n - 1$th letter to create b_2. Then, b_2 starts with the unique permutation

$$(x_2, x_3, \ldots, x_{n-1}, x_1, x_n).$$

Thus, the maximum possible intersection length between b_1 and b_2 is $\ell = n - 2$, since the letters x_2 through x_{n-1} are shared.

(ii) *We require $(n-1)! - 1$ intersections in a superpermutation for n.*
Thinking of beads as nodes and intersections as the segments that connect them, each new node needs one new segment to connect to the existing structure. However, the first node does not need to be connected as it is the start of the structure. Thus, there are $(n-1)! - 1$ intersections: one less than the number of beads

□

From the result of part (i) of the preceding theorem, we create the first rule for this framework.

Rule 1 (Bead Transition Process) When transitioning from a bead b_i to b_{i+1}, set $b_{i+1} = \mathrm{SS}_{n-1}(b_i)$ (i.e. $k = 1$ in Definition 5).

Thus, this rule furnishes transitions from one bead to another while guaranteeing the maximum intersection length, a crucial feature that will be utilized in the construction of higher-level structures.

3.3 1-rings

Now, we introduce the 1-ring, a structure that contains beads as substructures, building our way up to the entire superpermutation in the process.

Definition 7 [1-ring] A 1-ring is the ordered cycle

$$R^1 = (b_1, b_2, \ldots, b_{n-1}),$$

where b_1 is any bead and every later entry is the same bead straight-shifted by SS_{n-1}:

$$b_j = \mathrm{SS}_{n-1}^{j-1}(b_1) \text{ for } 1 < j \leq n-1.$$

Here, indices are taken modulo $n-1$; after b_{n-1}, the cycle returns to b_1. We denote the set of all 1-rings as $\mathcal{R}^1$.

The following theorem will allow us to relate our previously defined straight-shift function to an incrementally larger structure.

Theorem 3 (1-ring Properties) The following properties hold for any 1-ring R_i^1:

(i) In transitioning from a 1-ring R_i^1 to R_{i+1}^1, the greatest possible intersection length is $\ell = n-3$.

(ii) Each superpermutation for $n \geq 4$ requires $(n-2)!$ 1-rings.

Proof We prove each component separately.

(i) *In transitioning from a 1-ring R_i^1 to R_{i+1}^1, the greatest possible intersection length is $\ell = n-3$.*

Consider the bead

$$b_1 = (x_1, \ldots, x_n, x_1, \ldots, x_{n-1}).$$

Following Rule 1 $n-2$ times, the bead

$$b_{n-1} = (x_{n-1}, x_1, \ldots, x_{n-2}, x_n, \ldots, x_{n-2})$$

is reached, and this marks the end of R_1^1 (Definition 7).

If we apply Rule 1 once more (duplicate b_{n-1} and straight-shift at index $n-1$), we obtain

$$b_n = (x_1, \ldots, x_n, x_1, \ldots, x_{n-1}),$$

which is identical to the original bead b_1 in R_1^1. Hence any overlap of length $n-2$ between two 1-rings would force their leading beads to coincide, contradicting Definition 7. Thus, an intersection length of $\ell = n-2$ between distinct 1-rings is impossible.

Now, we will examine if an intersection of length $\ell = n-3$ is possible. In doing so, we will attempt to share the letters

$$(x_2, x_3, \ldots, x_{n-2}).$$

Since we have cycled through all $n-2$ distinct straight-shifts at index $n-1$ (see Rule 1), any further application would return to a previously seen bead. Thus, we instead straight-shift b_1 in R_1^1 at the $(n-2)$-th index to obtain b_1 in R_2^1. Straight-shifting as prescribed yields

$$(x_2, x_3, \ldots, x_{n-2}, x_1, x_{n-1}, x_n, \ldots, x_{n-1})$$

as b_1 of R_2^1, which has an intersection of length $\ell = n-3$ with R_1^1.

(ii) *Each superpermutation for $n \geq 4$ requires $(n-2)!$ 1-rings.*
Each superpermutation for n requires $(n-1)!$ beads (Theorem 1), and each 1-ring contains $n-1$ beads (Definition 7). Thus, we require

$$\frac{(n-1)!}{n-1} = (n-2)!$$

1-rings in the superpermutation. □

Remark 1 For $n = 3$, the concept of a 1-ring degenerates to a single bead, hence we restrict $n \geq 4$.

The next rule follows from the proof of Theorem 3(i).

Rule 2 (1-Ring Transition Process) Let $R_i^1 = (b_1, \ldots, b_{n-1})$ be a 1-ring, where each b_j is a bead. To form the next 1-ring R_{i+1}^1, we:

1. Apply SS_{n-2} to the leading bead b_1 of R_i^1, yielding a new bead $\tilde{b}_1 := \mathrm{SS}_{n-2}(b_1)$.
2. Populate the remaining beads of R_{i+1}^1 by repeatedly applying SS_{n-1} to $\tilde{b}_1$:

$$R_{i+1}^1 \;=\; \Big(\tilde{b}_1,\ \mathrm{SS}_{n-1}(\tilde{b}_1),\ \mathrm{SS}_{n-1}^2(\tilde{b}_1), \ldots, \mathrm{SS}_{n-1}^{n-2}(\tilde{b}_1)\Big).$$

(This yields $n-1$ beads altogether.) Thus, R_{i+1}^1 is a 1-ring whose leading bead is given by $\mathrm{SS}_{n-2}(b_1)$, and whose subsequent beads follow our usual 1-ring definition $\big(b_j = \mathrm{SS}_{n-1}^{j-1}(b_1)\big)$, but with b_1 replaced by $\tilde{b}_1$.

Note that this rule is logically coherent with both Theorem 3(i) and Theorem 2(i), as it ensures an intersection length of $\ell = n-3$ between any two 1-rings while ensuring that all beads within either 1-ring maintain an intersection length of $\ell = n-2$.

3.4 k-rings

Now, using a 1-ring as the base case, we inductively define higher orders of rings to construct the entire superpermutation structure.

Definition 8 (k-ring) A k-ring is the ordered cycle

$$R^k = (R_1^{k-1}, R_2^{k-1}, \ldots, R_{n-k}^{k-1}),$$

where R_1^{k-1} is any (k-1)-ring. We denote the set of all k-rings as $\mathcal{R}^k$.

Let $n \geq 3$. We define a k-ring for integers $1 \leq k \leq n-3$ as follows:

(a) *Base Case*
The base case for this inductive construction will be the 1-ring, which is defined as an ordered cycle of beads

$$R^1 = (b_1, b_2, \ldots, b_{n-1}),$$

where

$$b_j = \mathrm{SS}_{\mathrm{n-1}}^{\mathrm{j-1}}(b_1) \text{ for } 1 < j \leq n-1.$$

(As stated previously, this transformation wraps mod $n-1$). The transition norm for 1-rings will also be utilized as the base case for the inductive construction as well.

(b) *Transition between k-rings for $k \geq 2$*
Now, we define how to "transition" from one $(k-1)$-ring to another, which we will utilize to construct our k-ring. Denote the $(k-1)$-ring transition operator as $\mathrm{Trans}(\cdot)$.
Given a $(k-1)$-ring R_i^{k-1}, we descend to the leading bead b of the structure (obtained by recursively descending through the first $k-1$ subring) and straight shift it at index $n-(k-1)-1 = n-k$ to obtain a new anchor bead b'. Build a new $(k-1)$-ring from scratch through the same process that yielded R_i^{k-1} and call it R_{i+1}^{k-1}. Specifically, $R_{i+1}^{k-1} := \mathrm{Trans}(R_i^{k-1})$. Similar to 1-rings, the $\mathrm{Trans}(\cdot)$ operator wraps mod $n-k$, where we have a k-ring constituted of a cycle of $k-1$-rings.

(c) *k-ring from scratch*
Finally, for $k \geq 2$, to build a k-ring from scratch we first build the initial subring R_1^{k-1} from scratch. Then, for $2 \leq j \leq (n-k)$, define

$$R_j^{k-1} = \mathrm{Trans}(R_{j-1}^{k-1}),$$

where each step uses the transition to go from the previous subring to the next. Finally, set

$$R^k := (R_1^{k-1}, R_2^{k-1}, \ldots, R_{n-k}^{k-1}),$$

completing the construction.

To clear up any ambiguity about this procedure, we illustrate the process for creating a 2-ring around the initial bead $b = (1,2,3,4,1,2,3)$ (here $n = 4$).

Example 1 (2-ring creation) To begin, we fall to our base case to create the first 1-ring in the cycle. To do so, we must obtain the ordered cycle

$$R_1^1 = (b, \mathrm{SS}_{n-1}(b), \mathrm{SS}_{n-1}^2(b)).$$

Performing the computation according to the definition of straight-shift yields

$$R_1^1 = ((1,2,3,4,1,2,3),(2,3,1,4,2,3,1),(3,1,2,4,3,1,2)).$$

Next, we must transition from R_1^1 to R_2^1. Following the defined transition rule (Rule 2) yields a new leading bead

$$b' = (2,1,3,4,2,1,3)$$

for R_2^1. Following the same procedure as R_1^1, we obtain

$$R_2^1 = ((2,1,3,4,2,1,3),(1,3,2,4,1,3,2),(3,2,1,4,3,2,1)).$$

Finally, we put these 1-rings together to obtain the final 2-ring:

$$R^2 = \Big(\Big((1,2,3,4,1,2,3),\ (2,3,1,4,2,3,1),\ (3,1,2,4,3,1,2)\Big),$$
$$\Big((2,1,3,4,2,1,3),\ (1,3,2,4,1,3,2),\ (3,2,1,4,3,2,1)\Big)\Big).$$

Note that any creation of a k-ring (where $k \geq 2$) would function similarly, in that it is constructed from descending to a constituent 1-ring and constructing around it as prescribed from Definition 8.

Now, we introduce a new notation that will be required to prove properties about k-rings.

Notation 2 (Abuse of Notation and Overlap Merging) Throughout this paper, we may sometimes denote a k-ring R^k as an ordered cycle of its constituent $(k-1)$-rings (or beads), or alternatively as a single flattened sequence in which any overlapping segments are merged. For example, in the case of a 1-ring R^1 over the set $\{1, 2, 3, 4\}$, one might write

$$R^1 = \big((1, 2, 3, 4, 1, 2, 3), (2, 3, 1, 4, 2, 3, 1), (3, 1, 2, 4, 3, 1, 2)\big)$$

to emphasize its structure as a cycle of three beads. However, for brevity, the same ring can also be represented by the 'flattened' sequence

$$(1, 2, 3, 4, 1, 2, 3,\ 1, 4, 2, 3, 1, 2, 4, 3, 1, 2),$$

where internal commas and parentheses are omitted and the overlapping segments shared between consecutive rings (or beads) are merged.

From this point on, the 'abuse' of merging overlaps in a flattened representation will be utilized whenever it is more convenient, with all details provided so there is no ambiguity.

For convenience, we add another name to identify beads.

Notation 3 (0-ring) A bead, b, can be equivalently denoted as a 0-ring, R^0.

We also formalize the notion of a 'leading bead' that was utilized in defining a k-ring.

Definition 9 (Leading bead) The leading bead of any R^k is the first R^0 in the sequence.

Now, we can prove desired properties about k-rings.

Theorem 4 (k-ring Properties) The following properties hold for all k-rings:

(i) We require $(n-(k+1))!$ k-rings in a superpermutation.
(ii) Any k-ring begins and ends with the same $n-(k+1)$ letters.
(iii) The maximum intersection length between two k-rings is $\ell = (n-(k+2))$.

Proof Each of these components will be proven separately.

(i) *We require* $(n-(k+1))!$ *k-rings in a superpermutation*
We will prove this via induction for a given n. As the base case, we know that we require $(n-1)!$ 0-rings (Theorem 1).

Thus, our inductive hypothesis is as follows: we require $(n-(k+1))!$ k-rings. Now, for the inductive step, we must prove that we require $(n-(k+2))!$ $(k+1)$-rings. Using our inductive hypothesis, we can divide the required number of k-rings by the amount of k-rings in each $(k+1)$-ring (Definition 8). This yields

$$\frac{(n-(k+1))!}{n-(k+1)} = (n-(k+2))!,$$

thus proving through induction for all $0 \leq k \leq n-3$.

(ii) *Any k-ring begins and ends with the same $n-(k+1)$ letters.*
We will prove this via induction for a given n. As the base case, we know that any 0-ring begins and ends with the same $n-1$ letters (Theorem 1).

Our inductive hypothesis is as follows: any k-ring begins and ends with the same $n-(k+1)$ letters. Now, for the inductive step, we must prove that any $k+1$-ring begins and ends with the same $n-(k+2)$ letters.

Suppose the leading bead of R^{k+1} is

$$b = (x_1, x_2, \ldots, x_n, \ldots, x_{n-1}).$$

We will now determine the leading bead of the last k-ring in this $k+1$-ring. We know there are $n-(k+1)$ k-rings in this $k+1$-ring (Definition 8), so we must perform $n-(k+1)-1 = n-k-2$ Straight-shifts at index $n-k-1$ to obtain this bead (recall that the leading bead of R^{k+1} is also the leading bead of R_1^k).

Applying this transformation yields the leading bead of R_{n-k-1}^k:

$$\tilde{b} = (x_{n-k-1}, x_1, x_2, \ldots, x_{n-k-2}, x_{n-k}, x_{n-k+1}, \ldots, x_n, \ldots, x_{n-1})$$

From the inductive hypothesis, we know that R_{n-k-1}^k will begin and end with the same $n-k-1$ letters (here we utilize the aforementioned flattened notation):

$$R_{n-k-1}^k = (x_{n-k-1}, x_1, x_2, \ldots, x_{n-k-2}, \ldots\ldots, x_{n-k-1}, x_1, x_2, \ldots, x_{n-k-2}).$$

As such, we see that R^{k+1} begins and ends with the same

$$n-k-2 = n-(k+2)$$

letters as desired, thus proving through induction for all $0 \leq k \leq n-3$.

(iii) *The maximum intersection length between two k-rings is $\ell = (n-(k+2))$.*
We will proceed via strong induction. For our base case, we know that 0-rings intersect with length $\ell = n-2$ (Theorem 2).

For the inductive hypothesis, assume that all j-rings, such that $0 \leq j \leq k$ intersect with length $\ell = n-(k+2)$. Furthermore, we utilize the definition of transitioning between k-rings (Definition 8(b)).

From (ii), we write

$$R_1^{k+1} = (x_1, x_2, \ldots, x_{n-k-2}, \ldots\ldots, x_1, x_2, \ldots, x_{n-k-2}).$$

Observe that we have already exhausted all possible permutations starting with $(x_1, x_2, \ldots, x_{n-k-2})$ due to the Straight-shift process at every mth letter, such that $n-k-3 < m \leq n-1$, performed by transitioning through all j-rings, such that $0 \leq j \leq k$.

Thus, R_2^{k+1} can, at most, share the letters $(x_2, x_3, \ldots, x_{n-k-2})$ with R_1^{k+1}. As before, we duplicate the leading bead of R_1^{k+1} and straight-shift at the $n-(k+1)-1$th letter. In doing so, R_2^{k+1} starts with the unique permutation

$$(x_2, x_3, \ldots, x_{n-k-2}, x_1, x_{n-k-1}, \ldots, x_n),$$

establishing intersections of length $\ell = n - (k+3)$ for $k+1$-rings, as desired. Thus, the proof is completed via strong induction for all $0 \leq k \leq n-3$. $\square$

Now, we formalize the rule that we utilized for the definition and proofs in the same format as previous orders of rings for future reference.

Rule 3 (k-ring Transition Process) Let $R_i^k = (R_1^{k-1}, \ldots, R_{n-k}^{k-1})$ be a k-ring, where each R_j^{k-1} is a $(k-1)$-ring. To form the next k-ring R_{i+1}^k, we:

1. Apply SS_{n-k-1} to the leading bead b of R_i^k, yielding a new bead $\tilde{b} := \mathrm{SS}_{n-k-1}(b)$.
2. Populate the remaining $(k-1)$-rings in the ordered cycle by using $\tilde{b}$ as the anchor bead to create R_{i+1}^k from scratch

3.5 The Structure

The following tree diagram illustrates the relationship between the various ring orders for a given n and its associated superpermutation W_n. The depicted relationship continues indefinitely while $n - k \geq 0$.

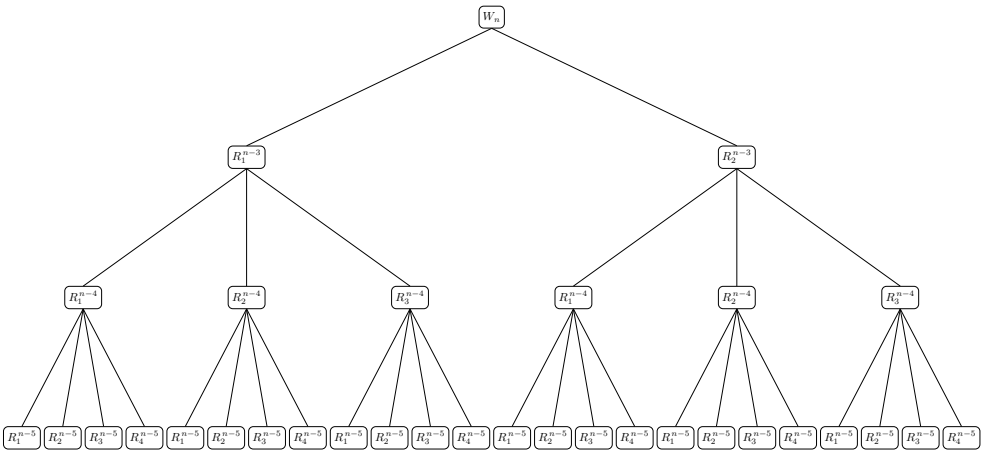

Fig. 3 Hierarchical relationship between k-rings.

The following definitions will be utilized to prove the next theorem.

Definition 10 (Mirroring Operation) Let s be a finite string indexed from 1 to ℓ. We define its *mirroring operation* as follows:

- **Case 1: $\ell = 2n+1$ is odd.** Let $i = n+1$ be the central index. For any index $m \in \{1, 2, \ldots, \ell\}$, the mirroring operation maps the character at index m to the position
$$i + (i - m).$$

In other words, we reflect each character across the central index i.

- **Case 2: $\ell = 2n$ is even.** Let the two central indices be $i = n$ and $j = n+1$. Then:
  - If $m \leq i$, the character at index m is mapped to position
  $$i + (j - m).$$
  - If $m \geq j$, the character at index m is mapped to position
  $$j + (i - m).$$

To quickly reference this operation, we introduce the following notation.

Notation 4 (Mirror Notation) The mirror of some string s is denoted as s^*.

Using this new notion of a mirror, we introduce the following functions to aid in more efficient construction of the superpermutation, as will be proven later.

Definition 11 (Mirror-shift MS_{n-k}) Let $1 \leq k \leq n-2$ and $R^0 = (x_1, \ldots, x_{2n-1}) \in \mathcal{R}^0$. Define $y = \mathrm{MS}_{n-k}(R^0)$ as:

$$y_m = \begin{cases} x_{m+k+1}, & 1 \leq m \leq n-k-2, \\ x_{n-m}, & n-k-2 < m < n, \\ x_m, & m = n, \\ y_{m-n}, & n < m \leq 2n-1. \end{cases}$$

Below, the mirror-shift function at the $n-1$th letter for the set $\{c_1, c_2, \ldots, c_5\}$ is depicted.

$$c_1c_2c_3c_4c_5c_1c_2c_3c_4 \longrightarrow c_3c_4c_2c_1c_5c_3c_4c_2c_1$$

Fig. 4 Application of mirror-shift MS_{n-1} on a bead over $\{c_1, c_2, \ldots, c_5\}$, where letters are inserted and mirrored about the $n-1$-th index; structural duplication is applied.

Definition 12 (Mirror-unshift MU_{n-k}) Let $1 \leq k \leq n-2$ and $R^0 = (x_1, \ldots, x_{2n-1}) \in \mathcal{R}^0$. Define $y = \mathrm{MU}_{n-k}(R^0)$ as:

$$y_m = \begin{cases} x_{n-m}, & 1 \leq m < k+2, \\ x_{m-(k+1)}, & k+2 \leq m < n, \\ x_m, & m = n, \\ y_{m-n}, & n < m \leq 2n-1. \end{cases}$$

Below, the mirror-unshift function at the $n-1$th letter for the set $\{c_1, c_2, \ldots, c_5\}$ is depicted.

$$c_1c_2c_3c_4c_5c_1c_2c_3c_4 \longrightarrow c_4c_3c_1c_2c_5c_4c_3c_1c_2$$

Fig. 5 Application of mirror-unshift MU_{n-1} on a bead over $\{c_1, c_2, \ldots, c_5\}$, where letters are re-inserted and mirrored about the $n-1$-th index; structural duplication is applied.

Similar to the formalized notion of a leading bead, we now introduce the concept of a trailing bead.

Definition 13 (Trailing bead) The trailing R^0 of some R^k is the last bead in that k-ring.

To prove the improved efficiency of these functions, we first must prove the following lemma and theorem.

Lemma 2 (Straight-shift Equivalence Lemma) Straight-shifting $j-2$ times at the $(j-1)$th index on some bead is equivalent to making the $j-1$-th letter the leading letter of that bead, moving each letter of index m (such that $1 \leq m \leq j-2$), to index $m+1$. Every letter at index m ($n < m \leq 2n-1$) is replaced by the letter at index $m-n$.

Proof Consider an arbitrary bead

$$R^0 = (x_1, x_2, \ldots, x_n, \ldots, x_{n-1}).$$

Select an arbitrary index $j-1$ as the straight-shift location. Applying $j-2$ straight-shifts yields the bead

$$\mathrm{SS}_{j-1}^{j-2}(R^0) = \bar{R^0} = (x_{j-1}, x_1, x_2, \ldots, x_{j-2}, x_j, x_{j+1}, \ldots, x_n, \ldots\ldots, x_{n-1}).$$

Hence, the letter formerly at index $j-1$ becomes the new leading letter and every letter at index m ($1 \leq m \leq j-2$) shifts to $m+1$, proving the claimed equivalence. □

Now, we utilize this result to prove the theorem.

Theorem 5 (Trailing bead Theorem) Assuming a leading bead $R^0 = (x_1, \ldots, x_n, \ldots, x_{n-1})$ within some k-ring, the trailing bead will be of the form

$$(x_{n-1}, x_{n-2}, \ldots, x_{n-k+1}, x_{n-k}, x_1, x_2, \ldots, x_{n-k-1}, x_n, \ldots\ldots, x_{n-k-1}).$$

Proof Consider some arbitrary k-ring R^k whose leading bead is equal to R^0 from the previous proof. To obtain the trailing bead, we must traverse through all R^j, such that $0 \leq j \leq k$ by performing the straight-shift operation at all respective indices. For each order, j, we will straight-shift at the $n-j-1$th letter (Rule 3), and we have $n-j-2$ available successive unique straight-shifts at this point.

To apply the straight-shifts for the $k-1$-rings, we use Lemma 2 to move the $n-k$th letter to the beginning of the string. Thus, we obtain the leading bead $\tilde{b}$ of R_{n-k}^{k-1}:

$$\tilde{b} = (x_{n-k}, x_1, x_2, \ldots, x_{n-k-1}, x_{n-k+1}, \ldots, x_n, \ldots\ldots, x_{n-1}).$$

Repeating this process for all subsequent j-rings ($j < k$), we obtain the trailing bead of R^k:

$$\bar{b} = (x_{n-1}, x_{n-2}, \ldots, x_{n-k+1}, x_{n-k}, x_1, x_2, \ldots, x_{n-k-1}, x_n, \ldots\ldots, x_{n-k-1})$$

Thus, the proof is complete. □

With this proof, we can now prove important properties about MU_{n-k} and MS_{n-k}. However, we first set the groundwork for function composition in this paper.

Notation 5 (Function Composition) The following notation will be utilized.

(i) The composition of any two functions is denoted by the symbol $\circ$.
(ii) Repeated composition of a function is denoted by $\bigcirc$. 'Regular' ordering (higher value on top and lower on bottom) indicates an orthodox output, akin to a summation. For example,

$$\bigcirc_{k=1}^{j} f_k = f_1 \circ f_2 \circ f_3 \circ \cdots \circ f_j.$$

Inverted ordering (lower value on top and higher on bottom) indicates that the output will run in reverse. For example,

$$\bigcirc_{k=j}^{1} f_k = f_j \circ f_{j-1} \circ f_{j-2} \circ \cdots \circ f_1.$$

Orthodox function composition norms apply (non-commutativity, right-to-left composition).

Now, we can establish the desired identities.

Theorem 6 (Function Identities) The following identities hold for all n, k.

(i) Mirror-unshift equals straight-unshift coupled with straight-shifts:

$$\mathrm{MU}_{n-k} = \Big(\bigcirc_{j=k+1}^{0} \bigcirc_{l=1}^{n-k-2+j} \mathrm{SS}_{n-k+j-2} \Big) \circ \mathrm{SU}_{n-k-1}.$$

(ii) Mirror-shift equals straight-unshifts coupled with straight-shift:

$$\mathrm{MS}_{n-k} = \mathrm{SS}_{n-k-1} \circ \bigcirc_{j=k+1}^{0} \bigcirc_{l=1}^{n-j-2} \mathrm{SU}_{n-j-1}.$$

Proof Each of these will be proven separately.

(i) *Mirror-unshift equals straight-unshift coupled with straight-shifts.*
First, we rewrite the double composition as the following equivalent expression:

$$\bigcirc_{l=1}^{n-2} \mathrm{SS}_{n-1} \circ \bigcirc_{l=1}^{n-3} \mathrm{SS}_{n-2} \circ \cdots \circ \bigcirc_{l=1}^{n-k-3} \mathrm{SS}_{n-k-2}.$$

Since l solely serves as an iterator, we can rewrite the expression once more:

$$\mathrm{SS}_{n-1}^{n-2} \circ \mathrm{SS}_{n-2}^{n-3} \circ \cdots \circ \mathrm{SS}_{n-k-2}^{n-k-3}.$$

Observe that this is simply an expression to obtain the trailing bead of some k-ring (Proof of Theorem 5).

Consider the two arbitrary k-rings R_i^k and R_{i+1}^k, with the leading bead of R_i^k being

$$R^0 = (x_1, x_2, \ldots, x_n, \ldots, x_{n-1}).$$

Thus, the leading bead of R_{i+1}^k is

$$\mathrm{SS}_{n-k-1}(R^0) = \tilde{R^0} = (x_2, x_3, \ldots, x_1, x_{n-k}, x_{n-k+1}, \ldots, x_n, \ldots, x_{n-1}).$$

Starting with $\tilde{R^0}$, we show that the two functions are equivalent.

Mirror-unshifting this bead yields

$$\mathrm{MU}_{n-k}(\tilde{R^0}) = \bar{R^0} = (x_{n-1}, x_{n-2}, \ldots, x_{n-k+1}, x_{n-k}, x_1, x_2, \ldots, x_{n-k-2}, x_{n-k-1}, x_n, \ldots, x_{n-1}).$$

Now, we apply the other transformation. Straight-unshifting yields

$$\mathrm{SU}_{n-k-1}(\tilde{R^0}) = R^0 = (x_1, x_2, \ldots, x_n, \ldots, x_{n-1}),$$

the leading bead of R_i^k. Now, by Theorem 5, the trailing bead of this k-ring is

$$R^{0'} = (x_{n-1}, x_{n-2}, \ldots, x_{n-k+1}, x_{n-k}, x_1, x_2, \ldots, x_{n-k-2}, x_{n-k-1}, x_n, \ldots, x_{n-1}).$$

Since $R^{0'} = \tilde{R^0}$, equivalence is proven.

(ii) *Mirror-shift equals straight-unshifts coupled with straight-shift.*
We rewrite the double composition as the following equivalent expression:

$$\bigcirc_{l=1}^{n-k-3} \mathrm{SU}_{n-k-2} \circ \bigcirc_{l=1}^{n-k-2} \mathrm{SU}_{n-k-1} \circ \cdots \circ \bigcirc_{l=1}^{n-2} \mathrm{SU}_{n-1}.$$

Since l solely serves as an iterator, we can rewrite the expression once more:

$$\mathrm{SU}_{n-k-2}^{n-k-3} \circ \mathrm{SU}_{n-k-1}^{n-k-2} \circ \cdots \circ \mathrm{SU}_{n-1}^{n-2}.$$

Observe that this is simply the reversal of (Proof of Theorem 5): an expression to obtain the leading bead of some k-ring from the trailing bead of that k-ring.

Consider the two arbitrary k-rings from part (i). Specifically, assume the trailing bead of R_i^k to be

$$\tilde{R^0} = (x_{n-1}, x_{n-2}, \ldots, x_{n-k+1}, x_{n-k}, x_1, x_2, \ldots, x_{n-k-2}, x_{n-k-1}, x_n, \ldots\ldots, x_{n-1}).$$

Applying this compound function to $\tilde{R^0}$ yields the leading bead of R_i^k:

$$\mathrm{SU}_{n-k-2}^{n-k-3} \circ \mathrm{SU}_{n-k-1}^{n-k-2} \circ \cdots \circ \mathrm{SU}_{n-1}^{n-2}(\tilde{R^0}) = R^0 = (x_1, x_2, \ldots, x_n, \ldots, x_{n-1}).$$

After straight-shifting R^0, we obtain the leading bead of R_{i+1}^k

$$\mathrm{SS}_{n-k-1}(R^0) = \bar{R^0} = (x_2, x_3, \ldots, x_1, x_{n-k}, x_{n-k+1}, \ldots, x_n, \ldots, x_{n-1}).$$

Conversely, mirror-shifting the trailing bead of R_i^k yields

$$\mathrm{MS}_{n-k}(\tilde{R^0}) = R^{0'} = (x_2, x_3, \ldots, x_1, x_{n-k}, x_{n-k+1}, \ldots, x_n, \ldots, x_{n-1}).$$

Since $R^{0'} = \tilde{R^0}$, equivalence is proven. □

Now, we can effectively transition between two k-rings without requiring the storage of a leading bead, since we can simply mirror-shift the trailing bead of any R_i^k to obtain the leading bead of R_{i+1}^k, enabling the construction from there.

Similarly, we can obtain the leading bead of any R_i^k by mirror-unshifting the trailing bead of R_{i+1}^k. This allows us to conclude that MS_{n-k} and MU_{n-k} are mutual inverses, an idea we formalize in the following corollary.

Corollary 1 (MS_{n-k} and MU_{n-k} are Inverses) Let $1 \leq k \leq n-2$. Then,

$$\mathrm{MS}_{n-k} = (\mathrm{MU}_{n-k})^{-1}$$

and

$$\mathrm{MU}_{n-k} = (\mathrm{MS}_{n-k})^{-1}$$

hold true.

Proof The result directly follows from the proof of Theorem 6, as we show that the compound functions are inverses of each other, and SS_{n-k} and SU_{n-k} are known inverses (Lemma 1). □

More importantly, these new functions facilitate a further reduction in space requirements, as we can simply store the current bead and mirror-shift it through all k-rings ($1 \leq k \leq n-3$) rather than storing the leading bead of every k-ring.

The following theorem furnishes an equivalent tradeoff between practical runtime and memory usage as a natural consequence of the structure that we generate via this procedure.

Theorem 7 (Palindromic Nature of the Resulting Superpermutation) R_2^{n-3} is the mirror of R_1^{n-3}.

Proof Consider the arbitrary $n-3$-ring R_1^{n-3} with leading bead $R^0 = (x_1, x_2, \ldots, x_n, \ldots, x_{n-1})$. We will first determine the trailing bead $\tilde{R^0}$ of R_1^{n-3}. Applying Theorem 5 yields:

$$\tilde{R^0} = (x_{n-1}, x_{n-2}, \ldots, x_3, x_1, x_2, x_n, \ldots, x_2).$$

To determine the leading bead of R_2^{n-3}, we simply take the leading bead of R_1^{n-3} and straight-shift at the 2nd character (Rule 3). Doing so yields:

$$\mathrm{SS}_2(R^0) = R^{0'} = (x_2, x_1, x_3, x_4, \ldots, x_{n-1}, x_n, \ldots, x_{n-1}).$$

As can be seen, these sequences are mirrors of each other. This will be used this as our base case.

Now, consider two arbitrary beads that are mirrors of each other. Without loss of generality, let

$$\bar{R^0} = (x_1, x_2, \ldots, x_n, \ldots\ldots, x_{n-1})$$

and

$$(\bar{R^0})^* = (x_{n-1}, x_{n-2}, \ldots, x_n, \ldots\ldots, x_1).$$

Consider the arbitrary index to mirror-unshift and mirror-shift to be the $n-k$th letter.

Mirror-unshifting $(\bar{R}^0)^*$ yields

$$\mathrm{MU}_{n-k}((\bar{R}^0)^*) = \hat{R}^0 = (x_1, x_2, \ldots, x_k, x_{k+1}, x_{n-1}, \ldots, x_{k+3}, x_{k+2}, x_n, \ldots, x_{k+2}),$$

and mirror-shifting $\bar{R}^0$ yields

$$\mathrm{MS}(\bar{R}^0) = \breve{R}^0 = (x_{k+2}, x_{k+3}, \ldots, x_{n-1}, x_{k+1}, x_k, \ldots, x_2, x_1, x_n, \ldots, x_1).$$

Thus, the mirror property is preserved after MU and MS for any arbitrary bead and its mirror, respectively, at any index.

In the superpermutation, since we start with two initial beads that are mirrors, any mirror-unshift or mirror-shift as we traverse through the first or latter half of the sequence, forward or backward, respectively, will preserve the mirror property. This ends with the leading bead $\breve{R}^0$ of R_1^{n-3}, being the mirror of the trailing bead $\hat{R}^0$ of R_2^{n-3}.

Specifically,

$$\breve{R}^0 = (x_1, x_2, \ldots, x_n, \ldots, x_{n-1})$$

and

$$\hat{R}^0 = (x_{n-1}, x_{n-2}, \ldots, x_2, x_1, x_n, \ldots, x_1),$$

where we derived $\hat{R}^0$ using Theorem 5 on the leading bead of R_2^{n-3}.

Thus,

$$R_2^{n-3} = (R_1^{n-3})^*,$$

making this structure a palindrome and completing the proof. □

4 The Algorithm

4.1 Mathematical Logic

To begin, we will outline the algorithm mathematically. Consider the + sign to denote concatenation rather than addition of numbers. The input is a set of n distinct letters:

$$s = \{x_1, x_2, x_3, \ldots, x_n\}.$$

From this input, we define our initial bead

$$b_o = (x_1, x_2, \ldots, x_n, \ldots\ldots, x_{n-1})$$

by concatenating s with all but the last letter of s.

We establish an integer ℓ to denote the intersection length, which is initially set to the value $n-2$.

Next, we define a function ϕ_{n-k}, which takes the output of c_{n-k} and outputs the substring from ℓ to the length of the string.

Each successive ϕ function takes the output bead of the previous ϕ function as its input bead, with the first call to the ϕ function using b_o as its input.

Thus, the superpermutation W_n can be constructed:

$$W_n = b_o + \sum_{x=0}^{1} \left(\sum_{y=0}^{2} \left(\sum_{z=0}^{3} \left(\cdots \sum_{\kappa=0}^{n-3} \left(\left(\sum_{\gamma=0}^{n-2} \phi_{n-1}|_{n-2} \times 1_{\gamma<n-2} \right) \right.\right.\right.\right.$$

$$+ \phi_{n-2}|_{n-3} \times 1_{\kappa<n-3}\Bigg) \cdots + \phi_4|_3 \times 1_{z<3}\Bigg) + \phi_3|_2 \times 1_{y<2}\Bigg)$$

$$+ \phi_2|_1 \times 1_{x<1}\Bigg),$$

where $1_{x<f}$ serves as an indicator function that ensures that ϕ is only performed when $x < f$, and $|_{\ell=k}$ serves as an indication of the intersection length to be considered by the ϕ function.

Now, we calculate the total number of operations.

Theorem 8 (Operation Count) The total number of operations in our algorithm for a given n is $(n-1)! - 1$.

Proof First, we define the function $T(d)$, which returns the number of operations at each level d, starting with $d = 2$ until $d = n$ (based on the number of iterations of the loops).

At each level, d, we:

- Perform d recursive calls.
- Call the ϕ function $d - 1$ times.

The edge case is where d=n, as the number of operations goes to 0 ($T(n) = 0$). Thus, we establish the recursive relation

$$T(d) = d \times T(d+1) + (d-1) \text{ for } 2 \leq d \leq n-1.$$

Now, for convenience, we define a new function

$$S(d) = T(d) + 1.$$

Thus,

$$T(d) = S(d) - 1.$$

Substituting this into the original equation and simplifying yields

$$S(d) = d \times S(d+1).$$

The boundary condition for T gives that

$$T(n) = 0 \implies S(n) = T(n) + 1 = 1.$$

Now, we expand the recurrence relation on S:

$$S(d) = d \times S(d+1) = d(d+1)S(d+2) = \prod_{j=d}^{n-1} j \times S(n).$$

Since $S(n) = 1$, it immediately follows that

$$S(d) = \prod_{j=d}^{n-1} j.$$

Expressed in factorial notation, this product is

$$S(d) = \frac{(n-1)!}{(d-1)!}.$$

Substituting T back in for S yields

$$T(d) = \frac{(n-1)!}{(d-1)!} - 1.$$

Finally, we are interested in the number of operations from $d = 2$. Substituting yields that there are

$$T(2) = (n-1)! - 1$$

total operations in our algorithm. □

In the subsequent subsection, we extend this procedure to provide a Java coded implementationof the algorithm.

4.2 Java Implementation

For the sake of simplicity, assume the n symbols are characters, entered into the program in array format.

We rewrite the iterative nested loops in a recursive fashion, having the function call itself at each iteration of a loop that runs a unique number of times based on our position in the recursive stack.

```
public void algo(int d) {
// When initially called, feed in d=2
      if (d>=n) {return;}
      for (int i = 0; i < d; i++) {
         algo(d+1);
         intersect = d-1;
         if (i == d-1) {return;}
         mirrorShift(d);
      }
   }
```

Now, we make optimizations that are only possible in code. First, we utilize Theorem 1(iii), by storing only the first n letters of a bead, as it saves time and space. Next, using efficient letter manipulation and pointers within the strings, we save time and space when performing the mirror-shift operation, as we do not need to store the entire bead, nor do we use time to mirror its substring.

Specifically, we only copy the portion of the bead that is to be mirrored, overwriting all other transformations on the existing structure. Rather than mirror the extracted substring, we assign a pointer to go from the end of the substring to the beginning, copying down letters to pre-determined indices.

On the following page, we provide an operational Java implementation of the program[1].

[1] Implementation Note: This program also returns the trivial superpermutations for $n = 1, 2$: 1 and 121, respectively.

```
public class SuperpermAlgo {

   private int n;
   private int intersect;
   private StringBuilder bead;
   private PrintWriter writer;

   public SuperpermAlgo(char[] c) {
      n = c.length;
      bead = new StringBuilder();
      try {writer = new PrintWriter("superperm_n" + n + ".txt");}
      catch (IOException e) {e.printStackTrace();}
      startBead(c);
      algo(2);
      writer.close();
   }

   public void startBead(char[] c) {
      for (int i = 0; i < n; i++) {
         bead.append(c[i]);
      }
      for (int i = 0; i < bead.length(); i++) {
         writer.print(bead.charAt(i));
      }
      for (int i = 0; i < bead.length()-1; i++) {
         writer.print(bead.charAt(i));
      }
   }

   public void algo(int d) {
      if (d>=n) {return;}
      for (int i = 0; i < d; i++) {
         algo(d+1);
         intersect = d-1;
         if (i == d-1) {return;}
         mirrorShift(d);
      }
   }

   public void mirrorShift(int index) {
      int k = n-index;
      String m = bead.substring(0, k);
      for (int i = 0; i < n-k-1; i++) {
         bead.setCharAt(i, bead.charAt(i+k));
      }
      for (int i = n-k-1; i < n-1; i++) {
         if (m.length() == 0) {break;}
         int l = m.length()-1-(i-(n-k-1));
         if (l < 0) {continue;}
         bead.setCharAt(i, m.charAt(l));
      }
      for (int i = intersect; i < bead.length(); i++) {
         writer.print(bead.charAt(i));
      }
      for (int i = 0; i < bead.length()-1; i++) {
         writer.print(bead.charAt(i));
      }
   }
}
```

The time and space complexity for this program will be formally outlined in a subsequent section.[2]

5 Determining the Length of the Sequence

Now, we determine the length of the sequences generated using this algorithm. To do so, we add the length of all beads and then subtract all intersections.

5.1 The Initial Count

Since there are $(n-1)!$ beads of length $2n-1$ in the superpermutation for n, the preliminary count will be

$$(n-1)! \times (2n-1).$$

5.2 The Intersections

Now, we will account for the intersections. We know that the intersection length between two k-rings is $\ell = n-k-2$. The following theorem will be utilized to determine the number of intersections between each order of ring.

Theorem 9 (Number of Intersections per ring Order) In a superpermutation for a given n, associated with k-rings, there are

$$i(k) = (n-(k+1))! - 1 - \sum_{j=k+1}^{n-3} i(j)$$

intersections of the same length.

Proof Firstly, we recall the nodes and edges argument from Theorem 2(ii). Using the same logic, we take the preliminary count of the number of intersections to be $(n-(k+1))! - 1$. However, there is one key difference in this application; this function is double-counting all intersections between j-rings, such that $k < j \leq n-3$, since all k-rings are contained in these j-rings.

Thus, the number of intersections between k-rings is recursively dependent on the sum of the number of intersections for all aforementioned j-rings, with $n-3$-rings being a base case, as there is precisely one intersection between the 2 rings of this order. □

5.2.1 Re-indexing the Recurrence

Corresponding to k-rings, the given intersection length is $\ell = n-k-2$. Thus, the original index k runs in the opposite direction to the lengths we care about: large values of k (near $n-3$) give the shortest intersections, while small values of k give the longest.

[2] The .java files can be found in the supplementary materials.

To align the index with the length itself, set $j = n-k-2$. Then, $j = 1$ corresponds to $k = n-3$ and intersections of length $\ell = 1$, and $j = n-2$ corresponds to $k = 0$ and to the maximal intersection of length $\ell = n-2$. Now, define

$$w_j = i(n-2-j),$$

so that w_j is both the number of intersections whose length equals j and also indexed by that same length.

Substituting $k = n-2-j$ into Theorem 9 yields the forward-running recurrence

$$w_j = (j+1)! - 1 - \sum_{m=0}^{j-1} w_m, \quad 1 \leq j \leq n-2,$$

with the understanding that $w_0 = 0$ (the 'intersections' of length 0 contribute nothing). This index choice lets j play a double role as both the loop variable and the intersection length, so later summations over j automatically multiply the correct length by the correct frequency.

5.2.2 Directly Defining the Recursive Relation

We will now find a closed form equation in terms of n for the value of any index of w. First, define the partial sum

$$S_j = \sum_{k=0}^{j} w_k.$$

Thus, we can see that

$$S_j = S_{j-1} + w_j.$$

From the original recursive relation, we know that

$$w_j = (j+1)! - 1 - S_{j-1}.$$

Next, we substitute this expression into the equation for S_j, yielding

$$S_j = S_{j-1} + ((j+1)! - 1 - S_{j-1}).$$

Simplifying, we obtain

$$S_j = (j+1)! - 1.$$

Now, w_j can be expressed as

$$w_j = S_j - S_{j-1}.$$

Inputting the calculated expressions for S_j and S_{j-1}, we obtain

$$w_j = ((j+1)! - 1) - (j! - 1).$$

Simplifying yields the following:

$$w_j = (j+1)! - j!,$$

$$w_j = (j+1)j! - j!,$$

$$w_j = ((j+1) - 1)j!,$$

and

$$w_j = j \cdot j!.$$

Thus, there are $j \cdot j!$ intersections of length j for $1 \leq j \leq n-2$. Observe that we can simplify this relation into the following summation:

$$\sum_{i=1}^{n-2} i(i \cdot i!).$$

Simplifying yields:

$$\sum_{i=1}^{n-2} i^2 \cdot i!.$$

5.3 Finalizing the Equation

Thus, the final equation is

$$l(n) = (2n-1) \times (n-1)! - \sum_{i=1}^{n-2} i^2 \cdot i!.$$

Remark 2 The closed-form length is equivalent to $\sum_{k=1}^{n} k!$. This numerical match with the (now-disproven) minimal-length conjecture is coincidental; our aim is $\mathcal{O}(n)$ working space, not length optimality (see [3] for the current best lower bound).

6 Time and Space Complexity Comparison of Methodologies

In this section, we provide a formal proof demonstrating that our mirror-shift algorithm is superior in terms of space efficiency compared to recursive and graph-theoretic approaches.

One assumption we will be making is that printing is equivalent to outputting in terms of time complexity, both being $\mathcal{O}(n)$ for a string of length n.

6.1 Overview of Other Methods

6.1.1 Recursive Methods

Recursive methods build superpermutations by [1, 2]:

- Starting from a base case (e.g., $k = 2$) and incrementally constructing superpermutations for $k+1$ by inserting the new symbol between two copies of every permutation of k, finally recombining all elements and eliminating overlap.
- Requiring the construction of superpermutations for all $k < n$ before constructing for n.

6.1.2 Graph-Theoretic Methods

Graph-theoretic approaches involve [1, 3, 4]:

- Modeling permutations as nodes in a graph, with edges representing overlaps between permutations.
- Seeking the shortest Hamiltonian path or walk that covers all permutations, which is an NP-hard problem.

6.2 Time and Space Complexity Analysis

6.2.1 Mirror-shift Method

Time Complexity

Theorem 10 (Time Complexity of the Mirror-shift Algorithm) The time complexity of the mirror-shift superpermutation construction method is $\mathcal{O}(n!)$.

Proof We proceed by analyzing the components of our method (assuming we are not utilizing mirroring)[3]:

1. **Initial Bead Generation**

   (i) Generating and printing the initial bead has a time complexity of $\mathcal{O}(n)$.

2. **Mirror-shift Operations**

   (i) The number of mirror-shifts is $(n-1)! - 1 = \mathcal{O}(n!)$.

   (ii) Each mirror-shift has an upper-bound time complexity of $\mathcal{O}(n)$.

   (a) Copying the portion of the bead to be mirrored onto another string has the upper-bound time complexity of $\mathcal{O}(n)$, since $k \leq n-3$ and we are mirroring $k+1$ letters.

   (b) Finally, overwriting the letters with their new transformations in the current bead has an upper-bound time complexity of $\mathcal{O}(n)$.[4]

[3] Recall that the proof of the palindromic nature of the structure generated by our mirror-shift methodology enables a practical runtime speedup via an equivalent tradeoff in memory usage.

[4] Rather than explicitly performing a mirroring operation, we can directly overwrite the bead with the new transformation by utilizing a pointer that goes from the end of the to-be-mirrored substring to the beginning rather than the other way around.

3. **Printing**

(i) Printing each bead has an upper-bound time complexity of $\mathcal{O}(n)$.

Therefore, the total time complexity is dominated by the mirror-shifts:

$$T(n) = \mathcal{O}(n! \cdot n) = \mathcal{O}(n!).$$

□

Space Complexity

Theorem 11 The space complexity of the mirror-shift method is $\mathcal{O}(n)$.

Proof Our method optimizes space via incremental output. beads are printed as they are generated, avoiding the need to store the entire superpermutation in memory during its construction.

The upper-bound storage requirements are as follows:

(i) The first n letters of the current bead: $\mathcal{O}(n)$.
(ii) The recursive call stack: $\mathcal{O}(n)$.
(iii) The portion of the bead being mirrored in the mirror-shift function: $\mathcal{O}(n)$.
(iv) A small number of variables for indices and positioning in generation: $\mathcal{O}(1)$.

Thus, the space complexity is asymptotically:

$$\mathcal{S}(n) = \mathcal{O}(n).$$

□

6.2.2 Recursive Methods

Time Complexity

Theorem 12 The time complexity of recursive methods for constructing superpermutations is $\mathcal{O}(n!)$.

Proof Recursive methods require constructing superpermutations for all k from 2 up to n:

$$T(n) = \sum_{k=2}^{n} T(k)$$

For each k:

$$T(k) = \mathcal{O}(k! \cdot k),$$

since:

(i) There are $(k-1)!$ permutations to consider.
(ii) Each permutation must be duplicated and appended to either end of k, with the identification taking $\mathcal{O}(k)$, concatenation is $\mathcal{O}(k)$, and concatenating to the superpermutation being constructed is $\mathcal{O}(k)$.
(iii) Eliminating overlaps in the resulting sequence has a time complexity of $\mathcal{O}(k)$ using the Knuth-Morris-Pratt pattern-finding algorithm.

(iv) Outputting the resulting string requires $\mathcal{O}(l)$ time, where l is the length of the string. A lower bound for this is $k!$.

Therefore:

$$T(n) = \sum_{k=2}^{n} k! = \mathcal{O}(n!)$$

Hence, the asymptotic time complexity is $\mathcal{O}(n!)$. □

Space Complexity

Theorem 13 The space complexity of recursive methods is $\mathcal{O}(n!)$.

Proof Recursive methods must store:

(i) The superpermutations for n and $n-1$, which has a lower bound space complexity of $\mathcal{O}(n!)$.
(ii) The current permutation being used, which has a space complexity of $\mathcal{O}(n)$.
(iii) The recursive call stack, which can reach a depth of n.
(iv) Auxiliary space for Knuth-Morris-Pratt and patternfinding: $\mathcal{O}(n)$

Therefore, the total space complexity is dominated by the storage of the superpermutations:

$$S(n) = \mathcal{O}(n!).$$

□

6.2.3 Graph-Theoretic Methods

Time Complexity

Theorem 14 The time complexity of graph-theoretic methods is $\mathcal{O}(2^{n!}n!^2)$ or $\mathcal{O}((n!)!)$.

Proof The components of Graph-theoretic methods are as follows:

- Graph Construction:

(i) Generating nodes, with each representing a unique permutation: $\mathcal{O}(n \cdot n!)$.
(ii) Generating an edge between each pair of nodes, with each edge representing the overlap between permutations (Knuth-Morris-Pratt): $\mathcal{O}(\binom{n!}{2} \cdot n)$.

- Hamiltonian Path Search (Two Approaches):

(i) Brute-forcing the shortest path has a time complexity of $\mathcal{O}(k!)$ for a graph containing k nodes: $\mathcal{O}((n!)!)$.
(ii) Using dynamic programming, approaches such as Held-Karp still require exponential time complexity; $\mathcal{O}(2^k \cdot k^2)$ for a graph containing k nodes: $\mathcal{O}(2^{n!}n!^2)$.

Therefore, the time complexity is dominated by the evaluation of the shortest Hamiltonian Path, with a best-case of

$$T(n) = \mathcal{O}(2^{n!}n!^2)$$

using Held-Karp, and a worst-case time complexity of

$$T(n) = \mathcal{O}((n!)!)$$

using brute force.

□

Space Complexity

Theorem 15 The space complexity of graph-theoretic methods is $\mathcal{O}((n!)^2)$.

Proof Graph-theoretic approaches must store:

(i) The nodes of the graph, where each represents one n-length permutation; for $n!$ permutations: $\mathcal{O}(n! \cdot n)$.
(ii) The edges of the graph, where there are $\binom{k}{2}$ for k nodes; for $n!$ nodes: $\mathcal{O}(\binom{n!}{2})$.
(iii) Auxiliary space for pathfinding algorithms; $\mathcal{O}(n \cdot 2^n)$ for Held-Karp, or $\mathcal{O}(n)$ for brute force.

Thus, the space complexity is dominated by the storage of the edges of the graph:[5]

$$S(n) = \mathcal{O}(n!^2).$$

□

6.3 Comparative Analysis

6.3.1 Time Complexity Analysis

Our methodology achieves minimal time complexity for constructing a superpermutation containing all $n!$ permutations. In practical applications, recursive methods have additional overhead due to cumulative construction for all $k < n$, and graph-theoretic methods suffer from the NP-hardness of finding the shortest Hamiltonian path.

6.3.2 Space Complexity Analysis

Our mirror-shift algorithm is highly space-efficient, requiring the storage proportional to half of a single bead (length n), the recursive stack, a small substring, and two variables. In contrast, recursive methods must store the superpermutations for n and $n-1$, and graph-theoretic methods require immense space to store the entire graph structure.

[5] Note that space optimization methods such as dynamic edge calculation can reduce the asymptotic space complexity to $\mathcal{O}(n!)$ with an equal tradeoff in time complexity. In this case, $\mathcal{O}(\binom{n!}{2} \cdot n)$ would be added to computation time.

References

[1] Houston, R.: Tackling the Minimal Superpermutation Problem (2014). https://arxiv.org/abs/1408.5108

[2] Johnston, N.: Non-uniqueness of minimal superpermutations. Discrete Mathematics **313**(14), 1553–1557 (2013) https://doi.org/10.1016/j.disc.2013.03.024

[3] Poster, A., Houston, R., Pantone, J., Vatter, V.: A lower bound on the length of the shortest superpattern (2018). https://oeis.org/A180632/a180632.pdf

[4] Williams, A.: Hamiltonicity of the Cayley Digraph on the Symmetric Group Generated by Sigma and Tau (2017). https://arxiv.org/abs/1307.2549

[5] Axler, S.: Linear Algebra Done Right (2024). https://linear.axler.net/LADR4e.pdf