An $O(n)$ Space Construction of Superpermutations

Dhruv Ajmera

TUMC 2025

What is a Superpermutation?

- A superpermutation is a sequence that contains every permutation of n symbols as a substring.

What is a Superpermutation?

- A superpermutation is a sequence that contains every permutation of n symbols as a substring.
- If we represent a permutation as a word, an example for $n = 3$ is $\boxed{123121321}$; the 3! permutations in order are $123, 231, 312, 213, 321$.

What is a Superpermutation?

- A superpermutation is a sequence that contains every permutation of n symbols as a substring.
- If we represent a permutation as a word, an example for $n = 3$ is $\boxed{123121321}$; the 3! permutations in order are $123, 231, 312, 213, 321$.
- Superpermutations are sometimes utilized for issues in encoding theory, cryptography, and sequence optimization.

Old Construction Methods

There are two superpermutation construction methods most prevalent in recent literature:

- Recursive Construction
- Least-Weight Hamiltonian Walk (TSP)

Recursive Construction

This construction algorithm is defined as follows.

Recursive Construction

This construction algorithm is defined as follows.

1. Take a superpermutation for n.

Recursive Construction

This construction algorithm is defined as follows.

1. Take a superpermutation for n.
2. Parse out the ith permutation in this superpermutation.

Recursive Construction

This construction algorithm is defined as follows.

1. Take a superpermutation for n.
2. Parse out the ith permutation in this superpermutation.
3. Insert the $n+1$th symbol between two copies of this permutation.

Recursive Construction

This construction algorithm is defined as follows.

1. Take a superpermutation for n.
2. Parse out the ith permutation in this superpermutation.
3. Insert the $n+1$th symbol between two copies of this permutation.
4. Append this string to the previously constructed string in the superpermutation for $n+1$.

Recursive Construction

This construction algorithm is defined as follows.

1. Take a superpermutation for n.
2. Parse out the ith permutation in this superpermutation.
3. Insert the $n+1$th symbol between two copies of this permutation.
4. Append this string to the previously constructed string in the superpermutation for $n+1$.
5. Eliminate any overlap.

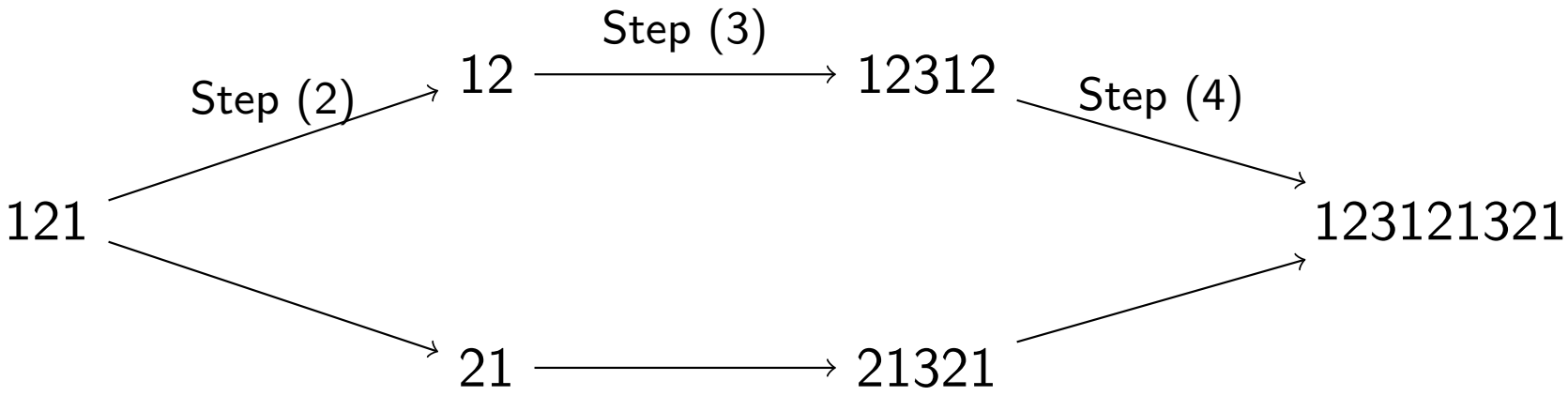
Recursive Construction

This construction algorithm is defined as follows.

1. Take a superpermutation for n.
2. Parse out the ith permutation in this superpermutation.
3. Insert the $n+1$th symbol between two copies of this permutation.
4. Append this string to the previously constructed string in the superpermutation for $n+1$.
5. Eliminate any overlap.

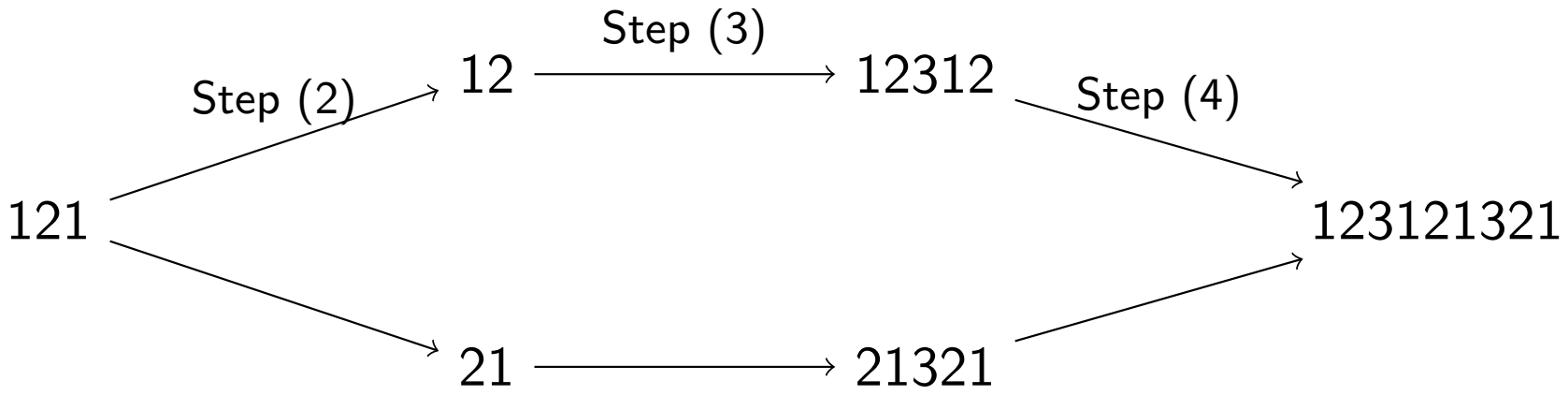
This process is repeated until all permutations from the superpermutation of n have been exhausted. As can be seen, this approach is factorial in both time and space. We show a visual representation of this process on the next slide.

Recursive Visual



As we can see, this procedure requires storing two intermediate sequences which are both factorial in size.

Recursive Visual



As we can see, this procedure requires storing two intermediate sequences which are both factorial in size.

The length of the superpermutation generated is

$$\sum_{k=1}^{n} k!.$$

Least-Weight Hamiltonian Walk (TSP)

This construction algorithm is defined as follows.

Least-Weight Hamiltonian Walk (TSP)

This construction algorithm is defined as follows.

1. Assemble a weighted graph where the nodes are every permutation of $[n]$.

Least-Weight Hamiltonian Walk (TSP)

This construction algorithm is defined as follows.

1. Assemble a weighted graph where the nodes are every permutation of $[n]$.
2. The weight of each directed edge is determined by $n -$ overlap (e.g. the overlap between 123 and 231 is 2, so the weight of the edge $123 \to 231$ is $3 - 2 = 1$).

Least-Weight Hamiltonian Walk (TSP)

This construction algorithm is defined as follows.

1. Assemble a weighted graph where the nodes are every permutation of $[n]$.
2. The weight of each directed edge is determined by $n -$ overlap (e.g. the overlap between 123 and 231 is 2, so the weight of the edge $123 \to 231$ is $3 - 2 = 1$).
3. Perform a least-weight Hamiltonian walk (traveling salesman problem).

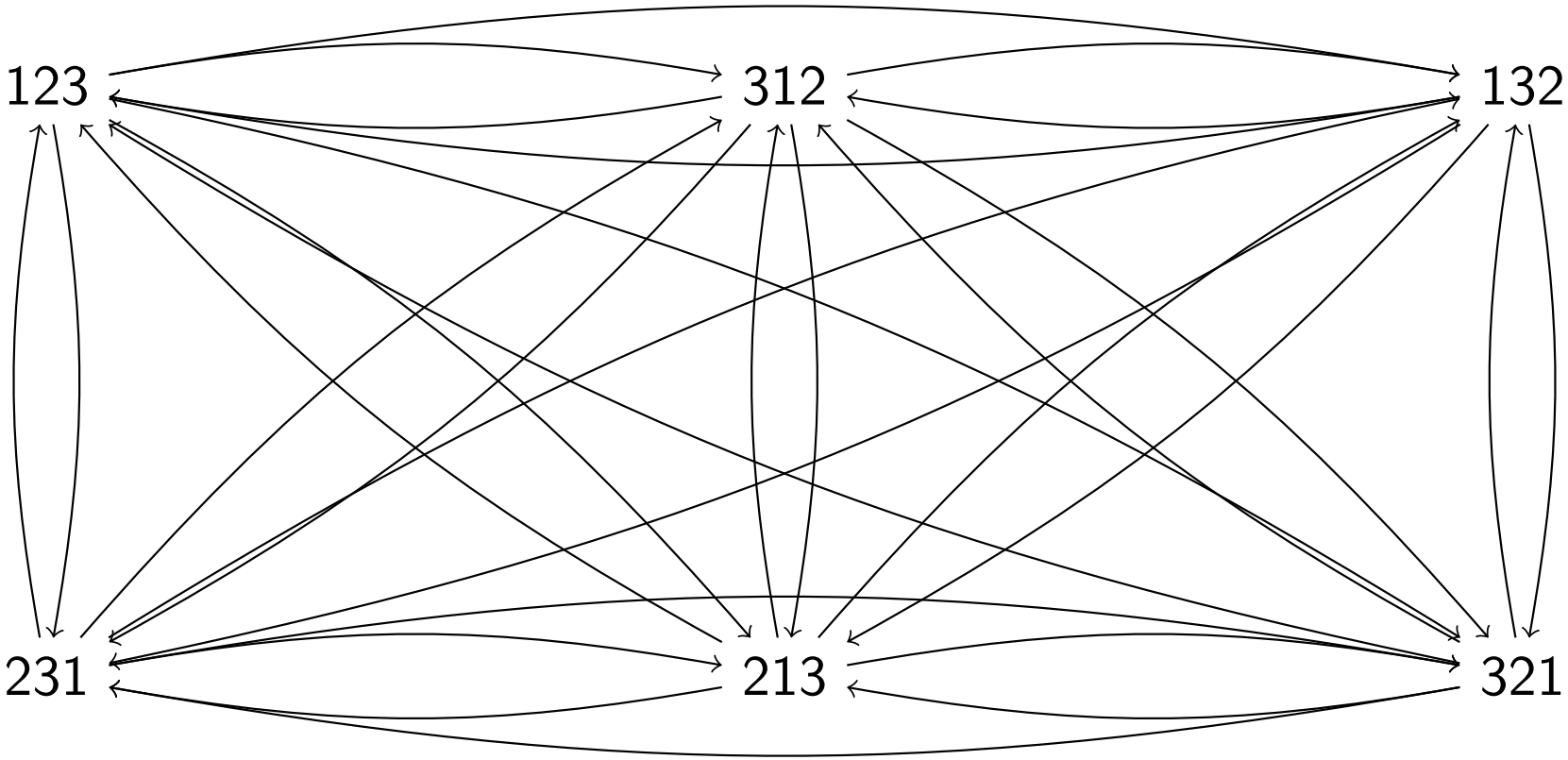
Least-Weight Hamiltonian Walk (TSP)

This construction algorithm is defined as follows.

1. Assemble a weighted graph where the nodes are every permutation of $[n]$.
2. The weight of each directed edge is determined by $n -$ overlap (e.g. the overlap between 123 and 231 is 2, so the weight of the edge $123 \to 231$ is $3 - 2 = 1$).
3. Perform a least-weight Hamiltonian walk (traveling salesman problem).

This is an incredibly computationally intensive endeavor, with time and space complexities far worse than factorial (in fact, attempting to create the shortest superpermutation for $n = 6$ via this method cost over 100 million CPU hours, with an optimistic time projection being over 50 years, with distributed computing!).

TSP Visual



For the sake of maintaining (some) clarity of this graph, we omit the weight labels for each edge. Since there are $n!$ nodes and $n!(n! - 1)$ edges, the space and time for such algorithms explodes extremely quickly, making them all but infeasible.

The Idea

What if, instead of trying to shave off a few extra characters from the final sequence, we try to make these sequences actually feasible to generate? (as feasible as possible, as the lower bound time complexity is $\mathcal{O}(n!)$).

The Bead-Ring Framework

We distill the main ideas of the framework and its logic here, omitting the proofs.

The Bead-Ring Framework

We distill the main ideas of the framework and its logic here, omitting the proofs.

If we let a *bead* b be a $2n-1$ tuple $(x_1, \ldots, x_{2n-1})$ that contains n permutations as contiguous subtuples (ex. $(1,2,3,1,2)$), then by repeatedly applying the function $MS_{n-k} : \mathcal{B} \mapsto \mathcal{B}$ (where $\mathcal{B}$ is the set of all beads) at different values of $k : 0 \leq k \leq n-3$, we obtain predictable intersections between beads (the intersection is defined as the number of shared indices between the end of a bead and the start of the next successive bead), creating the overall superpermutation.

The Bead-Ring Framework

We distill the main ideas of the framework and its logic here, omitting the proofs.

If we let a *bead* b be a $2n-1$ tuple $(x_1, \ldots, x_{2n-1})$ that contains n permutations as contiguous subtuples (ex. $(1,2,3,1,2)$), then by repeatedly applying the function $MS_{n-k} : \mathcal{B} \mapsto \mathcal{B}$ (where $\mathcal{B}$ is the set of all beads) at different values of $k : 0 \leq k \leq n-3$, we obtain predictable intersections between beads (the intersection is defined as the number of shared indices between the end of a bead and the start of the next successive bead), creating the overall superpermutation.

This function is defined as

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

Mirror-Shift Walkthrough

Here, we provide a step-by-step implementation by mirror-shifting the bead $b = (1, 2, 3, 4, 5, 1, 2, 3, 4)$ with the function MS_{n-1}.

Mirror-Shift Walkthrough

Here, we provide a step-by-step implementation by mirror-shifting the bead $b = (1, 2, 3, 4, 5, 1, 2, 3, 4)$ with the function MS_{n-1}.

1. We insert the first 2 characters behind the nth character, shifting down the others: $\implies (3, 4, 1, 2, 5, 1, 2, 3, 4)$.

Mirror-Shift Walkthrough

Here, we provide a step-by-step implementation by mirror-shifting the bead $b = (1, 2, 3, 4, 5, 1, 2, 3, 4)$ with the function MS_{n-1}.

1. We insert the first 2 characters behind the nth character, shifting down the others: $\implies (3, 4, 1, 2, 5, 1, 2, 3, 4)$.
2. Now, we mirror the inserted subtuple about the $n-1$th index: $\implies (3, 4, 2, 1, 5, 1, 2, 3, 4)$.

Mirror-Shift Walkthrough

Here, we provide a step-by-step implementation by mirror-shifting the bead $b = (1, 2, 3, 4, 5, 1, 2, 3, 4)$ with the function MS_{n-1}.

1. We insert the first 2 characters behind the nth character, shifting down the others: $\implies (3, 4, 1, 2, 5, 1, 2, 3, 4)$.
2. Now, we mirror the inserted subtuple about the $n-1$th index: $\implies (3, 4, 2, 1, 5, 1, 2, 3, 4)$.
3. Finally, each entry $y_m : m > n$ is equal to y_{m-n}: $b' = (3, 4, 2, 1, 5, 3, 4, 2, 1)$.

Bead-Ring Visual

We start with $b_0 = (1, 2, 3, 4, 1, 2, 3)$. Now,

Bead-Ring Visual

We start with $b_0 = (1, 2, 3, 4, 1, 2, 3)$. Now,

$$MS_n(b_0) = (2, 3, 1, 4, 2, 3, 1),$$

Bead-Ring Visual

We start with $b_0 = (1, 2, 3, 4, 1, 2, 3)$. Now,

$$MS_n(b_0) = (2, 3, 1, 4, 2, 3, 1),$$

$$MS_n^2(b_0) = (3, 1, 2, 4, 3, 1, 2),$$

Bead-Ring Visual

We start with $b_0 = (1, 2, 3, 4, 1, 2, 3)$. Now,

$$MS_n(b_0) = (2, 3, 1, 4, 2, 3, 1),$$

$$MS_n^2(b_0) = (3, 1, 2, 4, 3, 1, 2),$$

$$MS_{n-1} \circ MS_n^2(b_0) = (2, 1, 3, 4, 2, 1, 3),$$

Bead-Ring Visual

We start with $b_0 = (1, 2, 3, 4, 1, 2, 3)$. Now,

$$MS_n(b_0) = (2, 3, 1, 4, 2, 3, 1),$$

$$MS_n^2(b_0) = (3, 1, 2, 4, 3, 1, 2),$$

$$MS_{n-1} \circ MS_n^2(b_0) = (2, 1, 3, 4, 2, 1, 3),$$

$$MS_n \circ MS_{n-1} \circ MS_n^2(b_0) = (1, 3, 2, 4, 1, 3, 2),$$

Bead-Ring Visual

We start with $b_0 = (1, 2, 3, 4, 1, 2, 3)$. Now,

$$MS_n(b_0) = (2, 3, 1, 4, 2, 3, 1),$$

$$MS_n^2(b_0) = (3, 1, 2, 4, 3, 1, 2),$$

$$MS_{n-1} \circ MS_n^2(b_0) = (2, 1, 3, 4, 2, 1, 3),$$

$$MS_n \circ MS_{n-1} \circ MS_n^2(b_0) = (1, 3, 2, 4, 1, 3, 2),$$

$$MS_n^2 \circ MS_{n-1} \circ MS_n^2(b_0) = (3, 2, 1, 4, 3, 2, 1).$$

Bead-Ring Visual

We start with $b_0 = (1, 2, 3, 4, 1, 2, 3)$. Now,

$$MS_n(b_0) = (2, 3, 1, 4, 2, 3, 1),$$

$$MS_n^2(b_0) = (3, 1, 2, 4, 3, 1, 2),$$

$$MS_{n-1} \circ MS_n^2(b_0) = (2, 1, 3, 4, 2, 1, 3),$$

$$MS_n \circ MS_{n-1} \circ MS_n^2(b_0) = (1, 3, 2, 4, 1, 3, 2),$$

$$MS_n^2 \circ MS_{n-1} \circ MS_n^2(b_0) = (3, 2, 1, 4, 3, 2, 1).$$

(note that the specific multiplicities of function usage and index are all deterministic for any n). Now, if we flatten these tuples into words and remove any overlap (deterministic in the algorithm), we are left with the superpermutation for $n = 4$:

$$123412314231243121342132413214321$$

Significance

- This approach also generates superpermutations of length $\sum_{k=1}^{n} k!$.

Significance

- This approach also generates superpermutations of length $\sum_{k=1}^{n} k!$.
- In the actual coded implementation, we simply output this sequence incrementally since the intersection lengths are pre-determined, resulting in $\mathcal{O}(n)$ working space! (The practical runtime is also slightly more optimized).

Significance

- This approach also generates superpermutations of length $\sum_{k=1}^{n} k!$.
- In the actual coded implementation, we simply output this sequence incrementally since the intersection lengths are pre-determined, resulting in $\mathcal{O}(n)$ working space! (The practical runtime is also slightly more optimized).
- This is because we only need to store the current bead we are working with, making this problem (more) computationally feasible!

Significance

- This approach also generates superpermutations of length $\sum_{k=1}^{n} k!$.
- In the actual coded implementation, we simply output this sequence incrementally since the intersection lengths are pre-determined, resulting in $\mathcal{O}(n)$ working space! (The practical runtime is also slightly more optimized).
- This is because we only need to store the current bead we are working with, making this problem (more) computationally feasible!
- In fact, we produced the superpermutation for $n = 14$ on a consumer laptop (a text file of ~ 87 GB) in ~ 15 minutes! (This was accomplished using $<$ **1 MB of RAM** in stored intermediary data, ignoring Java overhead).

Significance (cont.)

Just for some sense of scale, we provide a comparison to the other construction techniques.

Significance (cont.)

Just for some sense of scale, we provide a comparison to the other construction techniques.

- For the recursive methodology, we would need $\sim$ **187 GB of RAM** (assuming 2 bytes per character in a String).

Significance (cont.)

Just for some sense of scale, we provide a comparison to the other construction techniques.

- For the recursive methodology, we would need $\sim$ **187 GB of RAM** (assuming 2 bytes per character in a String).
- For the graph-theoretic methodology, we would need $\sim$ **7.6 trillion GB of RAM** (with the optimistic assumption of 1 byte per entry in an adjacency matrix), or, more realistically, $\sim$ **91 trillion GB of RAM** (with the more realistic assumption of 12 bytes per entry in an adjacency list), which is larger than all available physical memory on Earth.

Code + Questions

```
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));
   }
}
```