A Unified Framework to Discover Permutation Generation Algorithms

Pramod Ganapathi Rezaul Chowdhury

State University of New York at Stony Brook, New York, USA
Email: {pramod.ganapathi,rezaul}@cs.stonybrook.edu

We present two simple, intuitive, and general algorithmic frameworks that can be used to design a wide variety of permutation generation algorithms. The frameworks can be used to produce 19 existing permutation algorithms, including the well-known algorithms of Heap, Wells, Langdon, Zaks, Tompkins, and Lipski. We use the frameworks to design two new sorting-based permutation generation algorithms, one of which is optimal.

Keywords: Permutation Frameworks; Permutation Algorithms; Permutation Generation

1. INTRODUCTION

Permutations is one of the most important topics in both combinatorics and computer science. Combinatorialists are interested in enumeration and analysis of combinatorial objects such as permutations and combinations. On the other hand, computer scientists are more interested in generation of such objects. Permutation generation is one beautiful problem that has attracted the attention of many scientists for decades. In over fifty years, more than fifty algorithms have been developed to generate permutations. The search for more algorithms continues primarily due to the sheer interest in solving this problem.

Applications of Permutation Generation. There are many practical applications that require permutation generation. A few potential applications include: (*i*) *Software testing* [1]: to test the performance of an operating system by executing all permutations of a set of tasks; (*ii*) *Communication networks, cryptography, and network security* [2]: in error detection and correction codes that enable reliable delivery of digital data over unreliable communication channels; (*iii*) *Randomized algorithms*: to generate random permutations in coding theory and simulations; (*iv*) *Operations research*: in problems such as traveling salesperson problem (TSP); and (*v*) *Computer security*: to unlock a password using the brute-force attack.

Existing Permutation Algorithms. We consider algorithms that can generate all distinct $n!$ arrangements/permutations of a set of n distinct elements. Permutation generation algorithms can be classified into different categories based on how they generate the next permutation. Permutations are generated based on: (*i*) *Swaps* [3, 4, 5, 6, 7, 8]: where two elements are interchanged; (*ii*) *Adjacent swaps* [9, 10, 11, 12]: where two adjacent elements are interchanged; (*iii*) *Reversals* [13]: where a certain prefix or suffix of a permutation is reversed; (*iv*) *Counters* [14]: where counts of the elements are decremented and incremented; (*v*) *Rotations* [15, 2]: where a certain prefix or suffix of a permutation is left- or right-rotated; (*vi*) *Unranking* [13, 16]: where a number from 1 to $n!$ is mapped on to a permutation; and (*vii*) *Additions* [17]: where a number in the base-n system that represents a permutation is added with another number in the same base.

Some of the algorithms mentioned above are surveyed in [18, 1, 19, 20, 21]. In the recent three decades, the focus has been towards developing parallel algorithms to generate permutations [22, 23, 24, 25, 26] as well as coming up with permutation algorithms for a multiset [27, 28, 2].

Our Permutation Frameworks. Different permutation generation algorithms use different techniques to solve the problem. So, a natural question to ask is:

> Question
>
> Is it possible to design a unified framework that can be used to produce a wide variety of permutation generation algorithms?

A unified algorithmic framework relates different permutation algorithms. The generality of a unified framework helps in the deeper understanding of the similarities between various seemingly unrelated permutation algorithms. Furthermore, unification aids in the design of algorithms (see [29, 30]), both quantitatively and qualitatively.

In this paper, we present two ***unified algorithmic frameworks*** to design 19 existing permutation algorithms. The unified frameworks are based on two permutation sequences. We use the frameworks to design two new sorting-based permutation algorithms.

Existing Permutation Frameworks. Sedgewick [1], in his classic survey, analyzes different permutation algorithms and identifies two permutation sequences to generate permutations. He observes that the control structure of many permutation algorithms are similar ([1], Section 1) and calls it "factorial counting", which is the basis of our first framework. However, the paper does not give any explicit framework based on permutation sequences to discover permutation algorithms. Lipski [8] gives the first explicit scheme to produce a class of permutation algorithms based on swaps or interchanges. He gives 16 different algorithms based on swaps, that includes well-known algorithms of Heap [4] and Wells [3]. Knuth [31] gives an explicit framework or generic permutation generator ([31], Algorithm G in Section 7.2.1.2) to discover permutation algorithms. The framework, which is slightly complicated, uses the same idea as factorial counting. Both Sedgewick and Knuth arrive at their frameworks from the core idea that permutations of $p_1, \ldots, p_{i-1}$ must be generated before the increment of p_i for all $i \in [2, n]$. Frameworks that are used to generate a variety of combinatoral objects exist [14, 32] but it is unlikely that they can be used to generate a wide variety of permutation algorithms.

In our paper, we construct frameworks from the permutation sequences and not from the constraints of which elements will be permuted. Though the underlying idea between Sedgewick and Knuth's framework and our first framework is the same, the approach of constructing the frameworks is fundamentally different. Hence, our frameworks are simpler and more intuitive. Also, to the best of our knowledge, our second framework has not been studied in the literature.

Our Contributions. Our contributions are as follows:

(1) [***Permutation Frameworks (Section 3).***] We present two simple and unified algorithmic frameworks based on permutation sequences, which can be used to design a class of permutation algorithms. Using the frameworks we are able to produce 21 permutation algorithms including the well-known algorithms of Wells, Langdon, Zaks, Tompkins, Lipski, and Heap (probably the fastest permutation algorithm in practice).
(2) [***Permutation Algorithms (Section 4).***] We use the existing permutation sequences to discover two new permutation generation algorithms that are based on sorting. We prove their correctness. We show that the running time of one of the two algorithms is optimal (i.e., asymptotically fastest).

Organization of the Paper. The paper is organized as follows. In Section 2, we describe two existing permutation sequences that are used to generate permutations of a distinct set of elements and give both recursive and iterative algorithms to generate these permutation sequences. In Section 3, we present two generic algorithmic frameworks that use permutation sequences to produce a class of permutation algorithms. In Section 4, we present two new algorithms produced from the frameworks that use sorting to generate permutations of a set of distinct elements. We conclude in Section 5.

n	$\mathcal{L}_n$	$\mathcal{R}_n$
2	2	2
3	23232	33233
4	23232423232423232423232	44434443444244434443444

TABLE 1. A few initial values of permutation sequences $\mathcal{L}_n$ and $\mathcal{R}_n$.

2. PERMUTATION SEQUENCES

In this section, we define two existing permutation sequences, describe their properties, present recursive and iterative algorithms to generate these sequences, and prove the correctness of algorithms.

2.1. Definitions

A ***permutation sequence***, for a set of n distinct elements, is a sequence of $n! - 1$ integers (in the range $[2, n]$) that is used to generate all $n!$ permutations of the set. The recurrence relations of the two permutation sequences called left and right sequences, denoted by $\mathcal{L}_n$ and $\mathcal{R}_n$, respectively, are:

$$\mathcal{L}_n = \begin{cases} 2 & \text{if } n = 2, \\ \mathcal{L}_{n-1}(n\mathcal{L}_{n-1})^{n-1} & \text{if } n > 2. \end{cases}$$

$$\mathcal{R}_n = \begin{cases} 2 & \text{if } n = 2, \\ n^{n-1}u_1u_2\ldots u_{(n-1)!-1} & \text{if } n > 2. \end{cases}$$

where $u_i = \mathcal{R}_{n-1}[i]n^{n-1}$ for $i \in [1, (n-1)! - 1]$ and $\mathcal{R}_{n-1}[i]$ denotes the ith element of $\mathcal{R}_{n-1}$. We use the term a^b to mean that a is repeated b times. The first few values of $\mathcal{L}_n$ and $\mathcal{R}_n$ are shown in Table 1. The sequences $\mathcal{L}_n$ and $\mathcal{R}_n$ are of size $n! - 1$. These permutation sequences are not new. They have been known for decades [12, 1, 13, 18, 33].

2.2. Properties

We give bounds for the average value of the natural numbers in the two permutation sequences. Theorem 2.1 is due to Zaks. The two following theorems are helpful in analyzing the computational complexities of the permutation generation algorithms that use the corresponding permutation sequences.

THEOREM 2.1 (Zaks [13]: $\mathcal{L}_n$ average). *The average value of $\mathcal{L}_n$ is upper bounded by the Euler's number, e.*

Proof. Let $c(n,i)$ be the number of times the integer i occurs in $\mathcal{L}_n$. It is easy to see that $c(n,i) = n!(i-1)/i!$. The average value of $\mathcal{L}_n$ as n tends to infinity is

$$\lim_{n\to\infty} \frac{1}{n!-1} \sum_{i=2}^{n} i \cdot c(n,i)$$
$$= \lim_{n\to\infty} \frac{1}{n!-1} \sum_{i=2}^{n} \frac{n!}{(i-2)!} = \sum_{i=2}^{\infty} \frac{1}{(i-2)!} = e$$

□

Theorem 2.2 ($\mathcal{R}_n$ average). *The average value of $\mathcal{R}_n$ is lower bounded by $n - 1/(n-2)$ for $n \geq 3$.*

Proof. Let a_n be the average value of $\mathcal{R}_n$. From the definition of $\mathcal{R}_n$, we have $a_n = (a_{n-1} \cdot ((n-1)!-1) + n \cdot ((n!-1) - ((n-1)!-1)))/(n!-1)$. This leads to $a_n = n + x_n(a_{n-1} - n)$, where $x_n = ((n-1)!-1)/(n!-1)$.

We use mathematical induction to prove the theorem. Let $l_n = n - 1/(n-2)$ and $S(n)$ represent the predicate $a_n > l_n$. (a) Basis: For $n = 3$, $a_3 = 2.8 > l_3 = 2$. Hence, $S(3)$ is true. (b) Induction: Assume $S(n-1)$ is true. We need to prove $S(n)$. Consider $a_{n-1} > l_{n-1}$, which implies $a_{n-1} - n > -(n-2)/(n-3)$.

If $p, q < 0$ and $p > q$, then $pp' > qq'$, where $q' > p' > 0$. As LHS and RHS are negative and $1/n > x_n > 0$, we can multiply x_n on LHS and $1/n$ on RHS retaining the inequality.

$$x_n(a_{n-1} - n) > -\frac{1}{n}\left(\frac{n-2}{n-3}\right)$$
$$\implies n + x_n(a_{n-1} - n) > n - \frac{1}{n}\left(\frac{n-2}{n-3}\right)$$
$$\implies a_n > n - \frac{1}{n}\left(\frac{n-2}{n-3}\right)$$
$$\implies a_n > n - \frac{1}{n-2}\left(\frac{(n-2)^2}{n(n-3)}\right)$$
$$\implies a_n > n - \frac{1}{n-2}\left(1 - \frac{n-4}{n^2-3n}\right)$$
$$\implies a_n > n - \frac{1}{n-2} + \frac{1}{n-2}\left(\frac{n-4}{n^2-3n}\right)$$

The term $(n-4)/((n-2)(n^2-3n)) \geq 0$ for $n \geq 4$. Therefore, the inequality becomes $a_n > n - 1/(n-2)$, which implies $a_n > l_n$. Thus, $S(n)$ is true. □

2.3. Algorithms

The recursive and iterative algorithms to generate the two permutation sequences are given in Figure 1. Algorithm ISequenceL and Theorem 2.1 are due to Zaks [13] and have been incorporated here for completeness. We use the two recursive algorithms given in Figure 1 to construct two algorithmic frameworks in Section 3.

2.4. Proofs of Correctness

We give correctness proofs for the four algorithms to generate the two permutation sequences.

Theorem 2.3 (SequenceL correctness). *Algorithm* SequenceL *generates $\mathcal{L}_n$.*

Proof. We use mathematical induction to prove the theorem. Let $S(n)$ represent the statement that the algorithm generates $\mathcal{L}_n$.
Basis. For $n = 2$, $\mathcal{L}_2 = 2$. Hence, $S(2)$ is true.
Induction. Assume $S(n)$ is true. We need to prove $S(n+1)$. The algorithm starts by the invocation SequenceL$(n+1)$. We see that the function SequenceL(n) is called $n+1$ times and between every two successive calls the integer $n+1$ is generated once. Clearly, the permutation sequence generation follows the rule of $\mathcal{L}_{n+1} = \mathcal{L}_n((n+1)\mathcal{L}_n)^n$. Thus, $S(n+1)$ is true. □

Theorem 2.4 (SequenceR correctness). *Algorithm* SequenceR *generates $\mathcal{R}_n$.*

Proof. We use mathematical induction to prove the theorem. Let $S(n)$ represent the statement that the algorithm generates $\mathcal{R}_n$.
Basis. For $n = 2$, $\mathcal{R}_2 = 2$. Hence, $S(2)$ is true.
Induction. Assume $S(n)$ is true. We need to prove $S(n+1)$. The algorithm starts by the invocation SequenceR(2), which in turn calls SequenceR(3) and the process continues. The function SequenceR(n) calls SequenceR$(n+1)$ n times. In every call of SequenceR$(n+1)$, the integer $n+1$ will be printed n times. Also, by the assumption of $S(n)$, an integer of $\mathcal{R}_n$ will be printed between every two successive calls of SequenceR$(n+1)$. Clearly, the permutation sequence generation follows the rule of $\mathcal{R}_{n+1} = (n+1)^n u_1 u_2 \ldots u_{n!-1}$, where $u_i = \mathcal{R}_n[i](n+1)^n$ for $i \in [1, n!-1]$ and $\mathcal{R}_n[i]$ is the ith element of $\mathcal{R}_n$. Thus, $S(n+1)$ is true. □

Theorem 2.5 (Zaks [13]: ISequenceL correctness). *Algorithm* ISequenceL *generates $\mathcal{L}_n$.*

Proof. (simplified from Zaks [13].) We use mathematical induction to prove the theorem. Let $S(n)$ represent the statement that the algorithm generates $\mathcal{L}_n$ and when it stops, the configuration will be $k = n+1$, $next[i] = i+1$ for $i \in [2, n-1]$, $count[i] = 0$ for $i \in [3, n]$, and $count[n+1] = 1$.
Basis. For $n = 2$, $\mathcal{L}_2 = 2$ and final configuration is fine. Hence, $S(2)$ is true.
Induction. Assume $S(n)$ is true. We prove $S(n+1)$ explaining the steps from Algorithm ISequenceL and Table 2 (top).

Step 0: Initially, the *count* array will be empty.
Step 1: The algorithm generates $\mathcal{L}_n$ and configuration will be $k = n+1$, and $count[n+1] = 1$.
Step 2: The algorithm generates $\mathcal{L}_n(n+1)\mathcal{L}_n$ and configuration becomes $k = n+1$, and $count[n+1] = 2$.

SEQUENCEL(k)

Input: k; **Output:** $\mathcal{L}_n$
Invocation: SEQUENCEL(n)

```
1.  if k = 2 then
2.    visit k
3.    return
4.  SEQUENCEL(k − 1)
5.  for i ← k to 2 do
6.    visit k
7.    SEQUENCEL(k − 1)
```

SEQUENCER(k)

Input: k; **Output:** $\mathcal{R}_n$
Invocation: SEQUENCER(2)

```
1.  if k = n then
2.    for i ← k to 2 do
3.      visit k; return
4.  SEQUENCER(k + 1)
5.  for i ← k to 2 do
6.    visit k
7.    SEQUENCER(k + 1)
```

ISEQUENCEL()

Output: $\mathcal{L}_n$

```
1.  for i ← 1 to n + 1 do
2.    count[i] ← 0
3.    next[i] ← i + 1
4.  k ← 2
5.  while k ≤ n do
6.    visit k
7.    if k ≠ 2 then
8.      k ← 2
9.    else
10.     k ← next[2]
11.     next[2] ← 3
12.     count[k] ← count[k] + 1
13.     if count[k] = k − 1 then
14.       count[k] ← 0
15.       next[k − 1] ← next[k]
16.       next[k] ← k + 1
```

ISEQUENCER()

Output: $\mathcal{R}_n$

```
1.  for i ← 1 to n + 1 do
2.    count[i] ← 0
3.    next[i] ← i − 1
4.  k ← n
5.  while k ≤ 2 do
6.    if k ≠ n then
7.      visit k; k ← n
8.    else
9.      for i ← k to 2 do visit k
10.     k ← next[n]
11.     next[n] ← n − 1
12.     count[k] ← count[k] + 1
13.     if count[k] = k − 1 then
14.       count[k] ← 0
15.       next[k + 1] ← next[k]
16.       next[k] ← k − 1
```

FIGURE 1. Recursive and iterative algorithms to generate the two permutation sequences: $\mathcal{L}_n$ and $\mathcal{R}_n$.

Step	Output	k	$count[n+1]$	$count[n+2]$
0		2	0	0
1	$\mathcal{L}_n$	$n+1$	1	0
2	$\mathcal{L}_n(n+1)\mathcal{L}_n$	$n+1$	2	0
⋮	⋮	⋮	⋮	⋮
n-1	$\mathcal{L}_n((n+1)\mathcal{L}_n)^{n-2}$	$n+1$	$n-1$	0
n	$\mathcal{L}_n((n+1)\mathcal{L}_n)^{n-1}$	$n+1$	0	0
n+1	$\mathcal{L}_n((n+1)\mathcal{L}_n)^{n}$	$n+2$	0	1

Step	Output	k	$count[1]$
0		$n+1$	0
1	$(n+1)^n$	$\mathcal{R}_n[1]$	0
2	$(n+1)^n u_1$	$\mathcal{R}_n[2]$	0
⋮	⋮	⋮	⋮
$n!$-1	$(n+1)^n u_1 u_2 \dots u_{n!-2}$	$\mathcal{R}_n[n!-1]$	0
$n!$	$(n+1)^n u_1 u_2 \dots u_{n!-1}$	1	1

TABLE 2. Top: Steps in the proof of Theorem 2.5. Bottom: Steps in the proof of Theorem 2.6. Here, $u_i = \mathcal{R}_n[i](n+1)^n$ for $i \in [1, n!-1]$.

The process continues.
Step n: The algorithm generates $\mathcal{L}_n((n+1)\mathcal{L}_n)^{n-1}$ and when $count[n+1] = n$, due to line 13, the configuration will be $count[n+1] = 0$, and $next[n] = n+2$.
Step $n+1$: The algorithm outputs $\mathcal{L}_n((n+1)\mathcal{L}_n)^n$ and when n is generated for the last time, the configuration will be $count[n+2] = 1$ and $next[n] = n+1$. Thus, $S(n+1)$ is true. □

THEOREM 2.6 (ISEQUENCER correctness). *Algorithm* ISEQUENCER *generates* $\mathcal{R}_n$.

Proof. We use mathematical induction to prove the theorem. Let $S(n)$ represent the statement that the algorithm generates $\mathcal{R}_n$ and when it stops, the configuration will be $k = 1$, $next[i] = i-1$ for $i \in [1, n]$, $count[i] = 0$ for $i \in [2, n]$, and $count[1] = 1$.
Basis. For $n = 2$, $\mathcal{R}_2 = 2$ and final configuration is fine. Hence, $S(2)$ is true.
Induction. Assume $S(n)$ is true. We prove $S(n+1)$ explaining the steps from Algorithm ISEQUENCER and Table 2 (bottom).

Step 0: Initially, the *count* array will be empty.

Step 1: The algorithm generates $(n+1)^n$ and configuration will be $k = \mathcal{R}_n[1] = n$ and $count[k]$ will be incremented by 1.
Step 2: The algorithm generates $(n+1)^n\mathcal{R}_n[1](n+1)^n$. The configuration will be $k = \mathcal{R}_n[2]$ with an increment on $count[k]$. The algorithm continuously generates the integers of $\mathcal{R}_n$ between consecutive chunks of $(n+1)^n$. The process continues.
Step $n!$: The algorithm outputs $(n+1)^n u_1 u_2 \dots u_{n!-1}$, where $u_i = \mathcal{R}_n[i](n+1)^n$ for $i \in [1, n!-1]$ and $\mathcal{R}_n[i]$ is the ith element of $\mathcal{R}_n$. Due to our assumption that $S(n)$ is true the configuration becomes $k = 1$ from line 10, $next[n+1] = n$ from line 11, and $count[1] = 1$ from line 12. As $k = 1$, the control exits from the while loop and the algorithm stops. Thus, $S(n+1)$ is true. □

3. PERMUTATION FRAMEWORKS

In this section, we present two generic algorithmic frameworks that can be used to produce several permutation generation algorithms.

A permutation framework is a generic algorithmic structure that can be used to obtain a class of permutation generation algorithms. Figure 2 shows our two frameworks with an empty function Func. Both the frameworks invoke the Func function with the two parameters: k and i. The aim of the Func function is to give a meaning to the input parameters k and i and generate the next permutation from the current permutation and store it in the global n-sized array $\mathcal{P} = [p_1, p_2, \dots, p_n]$. We initialize $\mathcal{P}$ to $[1, 2, \dots, n]$ and visit this permutation. It is important to note that most real-world applications do not require printing permutations. Hence, in all our pseudocodes, we visit $\mathcal{P}$ which means that we generate or populate $\mathcal{P}$ without actually printing it.

Figure 2 gives 7 definitions of the Func function that lead to 8 different permutation algorithms. Figure 3 gives 16 definitions used by Lipski, which include definitions for producing Heap's (definition 9) and Wells's (definition 8) algorithms. By defining the function Func differently i.e., by giving different meanings to the natural number k and the loop parameter i, we end up with 21 different permutation generation algorithms.

Table 3 gives a summary of the permutation algorithms produced from the algorithmic frameworks using various Func functions. We can produce Zaks's algorithm [13] from FrameworkL by using Zaks's function i.e., Reverse to reverse the last k elements of the current permutation. We can produce Langdon's algorithm [1, 15] from FrameworkR by using Langdon's function i.e., a sequence of Rotates to right rotate the last j ($j \leftarrow n$ to k) elements of the current permutation. We can produce Tompkins-Paige's algorithm [1, 34] from FrameworkL by using Tompkins-Paige's function i.e., a sequence of Rotate's to right rotate the last j ($j \leftarrow 2$ to k) elements of the current permutation. Tompkins-Paige's function presented in this paper is different from that of the original algorithm but has the same core idea. The function definitions mentioned above do not make use of the parameter i. They can be easily modified to work on the first k (or j) elements of the current permutation rather of the last k (or j) elements due to *symmetry*.

We can produce Heap's [1, 8, 4] and Wells's algorithms [1, 8, 3] from FrameworkL by using functions that in turn use the Swap procedure to interchange two elements of the current permutation, one of them being the kth element of $\mathcal{P}$. Similar to Heap's and Wells's algorithms, we can produce 14 more permutation algorithms designed by Lipski [8] from FrameworkL using the Swap procedure. Refer to Figure 3 for Lipski's 16 definitions, which include definitions for producing Heap's and Wells's algorithms.

We show the versatility of the algorithmic frameworks by designing two new permutation algorithms: PermutationL and PermutationR using the two frameworks FrameworkL and FrameworkR, respectively. The Func function used by the frameworks is based on sorting and in turn uses the SortedNext function on the last k elements of the current permutation in $\mathcal{P}$ to get a new permutation. It is important to observe that the SortedNext function is the first function that works for both frameworks: FrameworkL and FrameworkR. The two permutation algorithms PermutationL and PermutationR and their SortedNext function are presented in Section 4.

4. PERMUTATION ALGORITHMS

In this section, we present two permutation generation algorithms based on permutation sequences and prove their correctness.

A permutation $\mathcal{P} = [p_1, p_2, \dots, p_n]$ is an arrangement of n distinct elements. For simplicity we assume that the set of elements for which we want to generate permutations is the set of natural numbers from 1 to n. The algorithms we are going to present in the subsequent sections uses a global array $\mathcal{P}$ to store permutations, which is initialized to $[1, 2, \dots, n]$. Any new permutation generated is stored in $\mathcal{P}$.

4.1. Algorithms

We present two permutation generation algorithms called PermutationL and PermutationR. The two algorithms are based on left and right permutation sequences: $\mathcal{L}_n$ and $\mathcal{R}_n$, respectively. This implies that they are based on FrameworkL and FrameworkR, respectively. The recursive algorithms for PermutationL and PermutationR are given in Figure 4. The iterative algorithms for PermutationL and PermutationR can be easily obtained from the iterative algorithms for generating $\mathcal{L}_n$ and $\mathcal{R}_n$, but for simplicity we consider only recursive algorithms.

FRAMEWORKL(k)

Input: k, global: n, $\mathcal{P} \leftarrow [1, 2, \ldots, n]$
Output: Permutations of $\mathcal{P}$
Invocation: FRAMEWORKL(n)

```
1. if k = 2 then
2.    FUNC(k, k)
3.    visit P
4. else
5.    FRAMEWORKL(k − 1)
6.    for i ← k to 2 do
7.       FUNC(k, i); visit P
8.       FRAMEWORKL(k − 1)
```

FRAMEWORKR(k)

Input: k, global: n, $\mathcal{P} \leftarrow [1, 2, \ldots, n]$
Output: Permutations of $\mathcal{P}$
Invocation: FRAMEWORKR(2)

```
1. if k = n then
2.    for i ← k to 2 do
3.       FUNC(k, i); visit P
4. else
5.    FRAMEWORKR(k + 1)
6.    for i ← k to 2 do
7.       FUNC(k, i); visit P
8.       FRAMEWORKR(k + 1)
```

FUNC(k, i)

Input: k, i, global: n, $\mathcal{P}$
Output: Next permutation of $\mathcal{P}$
{ Any function that works }

FUNC(k, i) ▷ Our function for FL and FR

```
1. SORTEDNEXT([p_{n−k+1}, p_{n−k+2}, ..., p_n])
```

FUNC(k, i) ▷ Zaks's function for FL

```
1. REVERSE([p_{n−k+1}, p_{n−k+2}, ..., p_n])
```

FUNC(k, i) ▷ Langdon's function for FR

```
1. for j ← n to k do
2.    ROTATE([p_{n−j+1}, p_{n−j+2}, ..., p_n])
```

FUNC(k, i) ▷ Tompkins's function for FL

```
1. for j ← 2 to k do
2.    ROTATE([p_{n−j+1}, p_{n−j+2}, ..., p_n])
```

FUNC(k, i) ▷ Heap's function for FL

```
1. j ← k − i + 1
2. if k % 2 = 0 then SWAP(p_k, p_j)
3. else SWAP(p_k, p_1)
```

FUNC(k, i) ▷ Wells's function for FL

```
1. j ← k − i + 1
2. if k % 2 = 0 and j > 2 then SWAP(p_k, p_{k−j})
3. else SWAP(p_k, p_{k−1})
```

FIGURE 2. (Top part) Our two unified algorithmic frameworks to design permutation generation algorithms using permutation sequences. (Remaining part) Definitions of the FUNC function used to produce several permutation generation algorithms. Here, FL = FRAMEWORKL and FR = FRAMEWORKR.

Algorithms	FUNC	FRAMEWORKL	FRAMEWORKR	Reference
Heap	SWAP	✓	–	[1, 8, 4]
Wells	SWAP	✓	–	[1, 8, 3]
Lipski's 14 algorithms	SWAP	✓	–	[8]
Zaks	REVERSE	✓	–	[13]
Tompkins	ROTATE	✓	–	[1, 34] 👍
Langdon	ROTATE	–	✓	[1, 15]
Our algorithms	SORTEDNEXT	✓	✓	👍

TABLE 3. FUNC functions for 21 permutation algorithms for the two frameworks: FRAMEWORKL and FRAMEWORKR, respectively. 👍 represents this paper.

The algorithm PERMUTATIONL basically generates the left permutation sequence $\mathcal{L}_n$, which is a sequence of $n! - 1$ natural numbers and sends each of these natural numbers, k, as an argument to the function FUNC. Each time the function FUNC is called it generates the next permutation from the current permutation using a special technique and stores the new permutation in $\mathcal{P}$. Thus, the algorithm PERMUTATIONL generates all the $n!$ permutations of $[1, 2, \ldots, n]$. In a similar way, PERMUTATIONR, too, generates all the $n!$ permutations of $[1, 2, \ldots, n]$ using the right permutation sequence $\mathcal{R}_n$. The PERMUTATIONR algorithm was designed by Anil Bhandary and the first author.

Both the algorithms PERMUTATIONL and PERMUTATIONR call the function FUNC with a parameter k that uses a special technique to generate the next permutation by reordering the last k elements of the current permutation $\mathcal{P}$ and not affecting the first $n-k$ elements of $\mathcal{P}$. The technique is called SORTEDNEXT, which is defined as follows.

DEFINITION 4.1 (**SortedNext**). *Let* $[a_1, a_2, \ldots, a_m]$

```
Func(k, i)                                   ▷ Lipski's function 1
1. j ← k − i + 1
2. if (k % 2 = 0 and j < k − 1) or k = 2 then
3.    Swap(p_k, p_j)
4. else Swap(p_k, p_2)
```

```
Func(k, i)                                   ▷ Lipski's function 2
1. j ← k − i + 1
2. if k % 2 = 0 and k > 2 then
3.    if j < k − 1 then Swap(p_k, p_j)
4.    else Swap(p_k, p_{k−2})
5. else Swap(p_k, p_{k−1})
```

```
Func(k, i)                                   ▷ Lipski's function 3
1. j ← k − i + 1
2. if k % 2 = 0 and k > 2 then
3.    if j > 1 then Swap(p_k, p_{k−j})
4.    else Swap(p_k, p_{k−2})
5. else Swap(p_k, p_{k−1})
```

```
Func(k, i)                                   ▷ Lipski's function 4
1. j ← k − i + 1
2. if k % 2 = 0 then
3.    if j < k − 3 or k = 2 then Swap(p_k, p_j)
4.    else Swap(p_k, p_{2k−4−j})
5. else Swap(p_k, p_{k−2})
```

```
Func(k, i)                                   ▷ Lipski's function 5
1. j ← k − i + 1
2. if k % 2 = 0 then
3.    if j > 3 or k = 2 then Swap(p_k, p_{k−j})
4.    else Swap(p_k, p_{k−4+j})
5. else Swap(p_k, p_{k−2})
```

```
Func(k, i)                                   ▷ Lipski's function 6
1. j ← k − i + 1
2. if k % 2 = 0 then
3.    if j = 1 or j = k − 1 then Swap(p_k, p_{k−j})
4.    else Swap(p_k, p_j)
5. else Swap(p_k, p_1)
```

```
Func(k, i)                                   ▷ Lipski's function 7
1. j ← k − i + 1
2. if k % 2 = 0 then
3.    if j = 1 or j = k − 1 then Swap(p_k, p_j)
4.    else Swap(p_k, p_{k−j})
5. else Swap(p_k, p_1)
```

```
Func(k, i)                                   ▷ Lipski's function 8
1. j ← k − i + 1
2. if k % 2 = 0 and j > 2 then Swap(p_k, p_{k−j})
3. else Swap(p_k, p_{k−1})
```

```
Func(k, i)                                   ▷ Lipski's function 9
1. j ← k − i + 1
2. if k % 2 = 0 then Swap(p_k, p_j)
3. else Swap(p_k, p_1)
```

```
Func(k, i)                                   ▷ Lipski's function 10
1. j ← k − i + 1
2. if k % 2 = 0 then Swap(p_k, p_j)
3. else Swap(p_k, p_{k−2})
```

```
Func(k, i)                                   ▷ Lipski's function 11
1. j ← k − i + 1
2. if k % 2 = 0 then
3.    Swap(p_k, p_{(k−3+j) % (k−1)+1})
4. else Swap(p_k, p_1)
```

```
Func(k, i)                                   ▷ Lipski's function 12
1. j ← k − i + 1
2. if k % 2 = 0 then
3.    Swap(p_k, p_{(2k−j−3) % (k−1)+1})
4. else Swap(p_k, p_{k−1})
```

```
Func(k, i)                                   ▷ Lipski's function 13
1. j ← k − i + 1
2. if k % 2 = 0 and j < k − 2 then Swap(p_k, p_j)
3. else Swap(p_k, p_{k−1})
```

```
Func(k, i)                                   ▷ Lipski's function 14
1. j ← k − i + 1
2. if k % 2 = 0 and j > 2 then Swap(p_k, p_{j−1})
3. else Swap(p_k, p_1)
```

```
Func(k, i)                                   ▷ Lipski's function 15
1. j ← k − i + 1
2. if k % 2 = 0 and j > 1 then Swap(p_k, p_{j−1})
3. else Swap(p_k, p_{k−1})
```

```
Func(k, i)                                   ▷ Lipski's function 16
1. j ← k − i + 1
2. if k % 2 = 0 then Swap(p_k, p_{k−j})
3. else Swap(p_k, p_1)
```

FIGURE 3. FrameworkL can be used to produce Swap-based permutation algorithms using the definitions of Lipski [8].

be a list of m distinct positive integers, and let $S = [s_1, s_2, \ldots, s_m]$ be those m elements in sorted order. Let $\textsc{Rank}(a_i)$*, where $i \in [1, m]$, represent the position of the element a_i in S i.e.,* $\textsc{Rank}(a_i) = j \in [1, m]$ *such that $a_i = s_j$. Then,* SortedNext *for $[a_1, a_2, \ldots, a_m]$ is defined as*

$$\textsc{SortedNext}([a_1, a_2, \ldots, a_m]) = [x_1, x_2, \ldots, x_m]$$
$$\text{such that } x_i = s_{1+(\textsc{Rank}(a_i) \bmod m)} \quad \forall i \in [1, m]$$

The recursion trees (or permutation trees) for generating permutations, are illustrated in Figure 5.

PERMUTATIONL(k)

Input: k, global: n, $\mathcal{P} \leftarrow [1, 2, \ldots, n]$
Output: Permutations of $\mathcal{P}$
Invocation: PERMUTATIONL(n)

```
1.  if k = 2 then
2.    FUNC(k)
3.    visit P
4.  else
5.    PERMUTATIONL(k − 1)
6.    for i ← k to 2 do
7.      FUNC(k); visit P
8.      PERMUTATIONL(k − 1)
```

PERMUTATIONR(k)

Input: k, global: n, $\mathcal{P} \leftarrow [1, 2, \ldots, n]$
Output: Permutations of $\mathcal{P}$
Invocation: PERMUTATIONR(2)

```
1.  if k = n then
2.    for i ← k to 2 do
3.      FUNC(k); visit P
4.  else
5.    PERMUTATIONR(k + 1)
6.    for i ← k to 2 do
7.      FUNC(k); visit P
8.      PERMUTATIONR(k + 1)
```

FUNC(k)

Input: k, global: n, $\mathcal{P}$
Output: Next permutation of $\mathcal{P}$

```
1.  { Apply SortedNext for the last k elements of P }
2.  [p_{n−k+1}, ..., p_n] ← SORTEDNEXT([p_{n−k+1}, ..., p_n])
```

FIGURE 4. Recursive permutation algorithms PERMUTATIONL and PERMUTATIONR to generate permutations of $\mathcal{P} = [1, 2, \ldots, n]$.

The leaf nodes of the trees represent the permutations. The nonleaf nodes of the trees represent the elements in the $\mathcal{L}_n$ and $\mathcal{R}_n$ permutation sequences. In the permutation tree corresponding to $\mathcal{L}_n$ sequence, when we go from bottom to top, the nonleaf nodes' values increase from 2 to n and the branch factor increases. In contrast, in the permutation tree corresponding to $\mathcal{R}_n$ sequence, when we go from bottom to top, the nonleaf nodes' values decrease from n to 2 and the branch factor decreases.

The process of generating the next permutation $\mathcal{B}$ from the current permutation $\mathcal{A}$ is as follows. Nodes $\mathcal{A}$ and $\mathcal{B}$ represent two consecutive leaves (or permutations) in the inorder traversal of the recursion tree. Let the node value of the least common ancestor (LCA) of consecutive leaves $\mathcal{A}$ and $\mathcal{B}$ be ℓ. Let FUNC(k) in Figure 4 be denoted by $\mathcal{F}_k$. Then, $\mathcal{F}_\ell$ is applied on permutation $\mathcal{A}$ to obtain permutation $\mathcal{B}$. Figure 5 depicts the recursion trees showing the permutations of $\mathcal{P} = [1, 2, 3]$ for PERMUTATIONL and PERMUTATIONR. Permutations of $\mathcal{P} = [1, 2, 3, 4]$ generated by PERMUTATIONL, PERMUTATIONR, Heap's, Wells's, Zaks's, Langdon's, and Tompkins's algorithms are shown in Figure 6.

4.2. Proofs of Correctness

We give correctness proofs for the two permutation generation algorithms. In our proofs we denote FUNC(k) by $\mathcal{F}_k$. Our algorithms work for any initial permutation of $[1, 2, \ldots, n]$.

THEOREM 4.1 (PERMUTATIONL correctness). PERMUTATIONL *generates all* $n!$ *unique permutations of* $[1, 2, \ldots, n]$.

Proof. We use induction to prove the theorem. Let $S(n)$ represent the predicate that PERMUTATIONL generates all $n!$ unique permutations of n elements using $\mathcal{L}_n$.
Basis. For $n = 2$, $\mathcal{L}_2 = 2$. Applying $\mathcal{F}_2$ on $[1, 2]$, we get $[2, 1]$. Hence, $S(2)$ is true.
Induction. Assume $S(n)$ is true. We need to prove $S(n + 1)$. The initial permutation is $[1, 2, \ldots, n + 1]$. We know that $\mathcal{L}_{n+1} = \mathcal{L}_n((n + 1)\mathcal{L}_n)^n$. By the assumption of $S(n)$, the algorithm generates all $n!$ unique permutations starting with 1 by permuting the last n elements of $[1, 2, \ldots, n + 1]$. Now, $\mathcal{F}_{n+1}$ is applied on the last permutation that starts with 1 to get a permutation starting with 2 because 2 is the next element of 1 in the sorted set $\{1, 2, \ldots, n + 1\}$. The algorithm generates all $n!$ unique permutations starting with 2 by permuting the remaining n elements. Again, $\mathcal{F}_{n+1}$ is applied on the last permutation to get a permutation that starts with 3 as 3 is the next element of 2 among all $n+1$ elements. This process continues for all $n + 1$ elements. At the end, the algorithm generates a total of $(n + 1)n! = (n + 1)!$ unique permutations. Thus, $S(n + 1)$ is true. □

THEOREM 4.2 (PERMUTATIONR correctness). PERMUTATIONR *generates all* $n!$ *unique permutations of* $[1, 2, \ldots, n]$.

Proof. We use induction to prove the theorem. Let $S(n)$ represent the predicate that PERMUTATIONR generates all $n!$ unique permutations of n elements using $\mathcal{R}_n$.
Basis. For $n = 2$, $\mathcal{R}_2 = 2$. Applying $\mathcal{F}_2$ on $[1, 2]$, we get $[2, 1]$. Hence, $S(2)$ is true.
Induction. Assume $S(n)$ is true. We need to prove $S(n + 1)$. The initial permutation is $[1, 2, \ldots, n + 1]$. We know that $\mathcal{R}_{n+1} = (n + 1)^n u_1 u_2 \ldots u_{n!-1}$, where $u_i = \mathcal{R}_n[i](n + 1)^n$ for $i \in [1, n! - 1]$ and $\mathcal{R}_n[i]$ is the

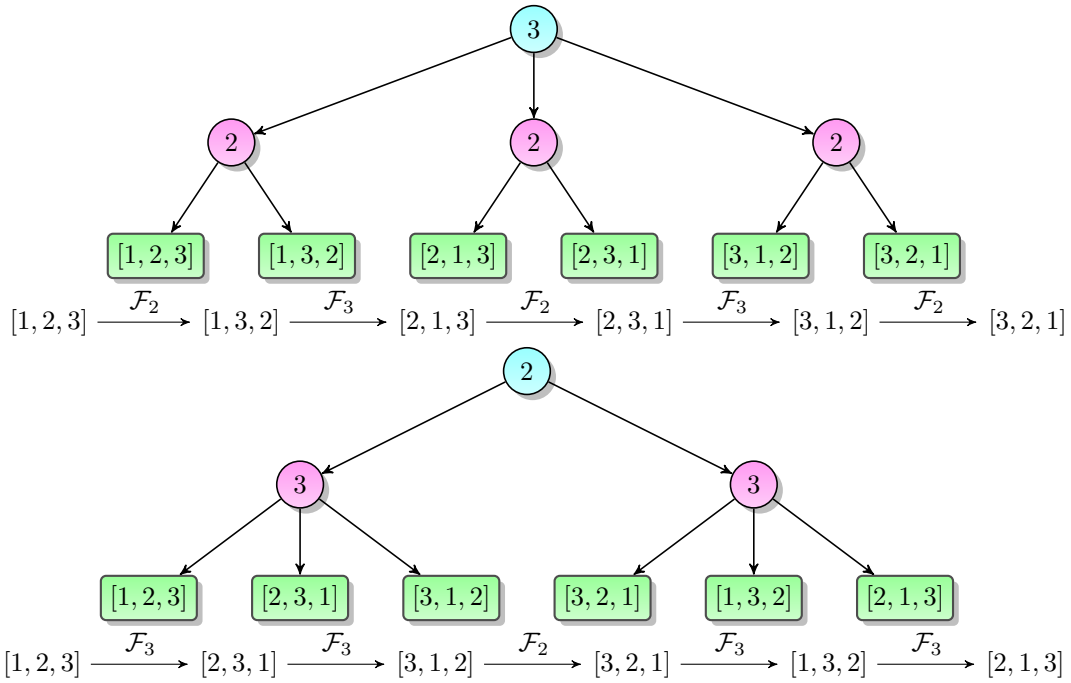

FIGURE 5. Permutation trees for PermutationL and PermutationR for $\mathcal{P} = [1, 2, 3]$.

ith element of $\mathcal{R}_n$. Divide the $(n+1)!$ permutations into $n!$ blocks, each block containing $n+1$ consecutive permutations. The first permutation of the first block is $[1, 2, \ldots, n+1]$. The algorithm applies $\mathcal{F}_{n+1}$ for n times to generate n new permutations. The second permutation starts with 2 as 2 is the next element of 1 among these $n+1$ elements, the third permutation starts with 3 as 3 is the next element of 2 and this continues. The first element of all $n+1$ permutations in the first block will be distinct. Similarly, the first element of all $n+1$ permutations in every block will be distinct. This implies that there will be a total of $n!$ permutations starting with 1 (one in every block), $n!$ permutations starting with 2, so on till element $n+1$.

Here we prove that the $n!$ permutations starting with an element i are distinct. Let $\mathcal{A}_i$ and $\mathcal{B}_i$ denote permutations starting with element i in some block k ($\in [1, n!-1]$) and $k+1$, respectively. Permutation $\mathcal{B}_i$ is obtained from $\mathcal{A}_i$ on successive applications of $\mathcal{F}_{n+1}$ for a times, then $\mathcal{F}_j$ ($j \leq n$ is in $\mathcal{R}_n$), and again $\mathcal{F}_{n+1}$ for b times such that $a + b = n + 1$. But, applying $\mathcal{F}_{n+1}$ a total of $n+1$ times is the same as not applying anything. This means we would have ended up with the same permutation $\mathcal{B}_i$ had we applied only $\mathcal{F}_j$ on $\mathcal{A}_i$.

$$\mathcal{A}_i \xrightarrow{\mathcal{F}_{n+1}^{a}\ \mathcal{F}_j\ \mathcal{F}_{n+1}^{n+1-a}} \mathcal{B}_i \quad \Longleftrightarrow \quad \mathcal{A}_i \xrightarrow{\mathcal{F}_j} \mathcal{B}_i$$

With this argument along with the assumption of $S(n)$ being true we see that the $n!$ permutations starting with an element i are indeed distinct. Hence, the algorithm generates a total of $(n+1)n! = (n+1)!$ unique permutations. Thus, $S(n+1)$ is true. □

4.3. Complexity Analysis

The complexity analysis remains the same for both recursive and iterative permutation algorithms. The space complexity of each of the two algorithms is $\Theta(n)$. The time complexity to print the permutations is $\Omega(n!n)$ for any permutation algorithm. However, most real-world applications don't require printing the permutations. They need to generate the permutations. Hence, we will only consider the time complexities (refer to [35], Section 1.7) to simply compute/generate/visit all the permutations, which are as follows:

(1) [PermutationL.] The algorithm uses the $\mathcal{L}_n$ permutation sequence. In the sequence $\mathcal{L}_n$, an element i occurs $c(n,i) = n!(i-1)/i!$ times. We can use any optimal comparison-sorting algorithm to compute SortedNext of a set of i elements. The total time required for all comparisons to generate permutations is computed as follows:

$$\begin{aligned}
\text{Time} &= \Theta\left(\sum_{i=2}^{n}\left(\begin{array}{l}\#\text{occurrences of } i \text{ in } \mathcal{L}_n \\ \times \text{ time for sorting } i\text{-sized array}\end{array}\right)\right) \\
&= \Theta\left(\sum_{i=2}^{n}(c(n,i)\cdot i\log i)\right) = \Theta\left(n!\sum_{i=2}^{n}\frac{\log i}{(i-2)!}\right) \\
&= \mathcal{O}\left(n!\sum_{i=2}^{\infty}\frac{\log i}{(i-2)!}\right) \\
&= \Theta(n!) \qquad \text{(convergence due to ratio test)}
\end{aligned}$$

The algorithm is ***optimal*** as the amortized cost of computing the next permutation is $\mathcal{O}(1)$.

What if we use hashing for SortedNext? Using

PL	PR	Heap	Wells	Zaks	Lang.	Tom.
[1234]	[1234]	[1234]	[1234]	[1234]	[1234]	[1234]
[1243]	[2341]	[2134]	[2134]	[1243]	[2341]	[1243]
[1324]	[3412]	[3124]	[2314]	[1342]	[3412]	[1342]
[1342]	[4123]	[1324]	[3214]	[1324]	[4123]	[1324]
[1423]	[4231]	[2314]	[3124]	[1423]	[1342]	[1423]
[1432]	[1342]	[3214]	[1324]	[1432]	[3421]	[1432]
[2143]	[2413]	[4213]	[1342]	[2341]	[4213]	[2341]
[2134]	[3124]	[2413]	[3142]	[2314]	[2134]	[2314]
[2341]	[3241]	[1423]	[3412]	[2413]	[1423]	[2413]
[2314]	[4312]	[4123]	[4312]	[2431]	[4231]	[2431]
[2431]	[1423]	[2143]	[4132]	[2134]	[2314]	[2134]
[2413]	[2134]	[1243]	[1432]	[2143]	[3142]	[2143]
[3124]	[2143]	[1342]	[1423]	[3412]	[1243]	[3412]
[3142]	[3214]	[3142]	[4123]	[3421]	[2431]	[3421]
[3214]	[4321]	[4132]	[4213]	[3124]	[4312]	[3124]
[3241]	[1432]	[1432]	[2413]	[3142]	[3124]	[3142]
[3412]	[1243]	[3412]	[2143]	[3241]	[1432]	[3241]
[3421]	[2314]	[4312]	[1243]	[3214]	[4321]	[3214]
[4132]	[3421]	[4321]	[3241]	[4123]	[3214]	[4123]
[4123]	[4132]	[3421]	[2341]	[4132]	[2143]	[4132]
[4231]	[4213]	[2431]	[2431]	[4231]	[1324]	[4231]
[4213]	[1324]	[4231]	[4231]	[4213]	[3241]	[4213]
[4321]	[2431]	[3241]	[4321]	[4312]	[2413]	[4312]
[4312]	[3142]	[2341]	[3421]	[4321]	[4132]	[4321]

FIGURE 6. Permutations of $\mathcal{P} = [1, 2, 3, 4]$ using different algorithms. The PERMUTATIONL and PERMUTATIONR algorithms are denoted by PL and PR, respectively.

most standard implementations of hashing to sort items in the PERMUTATIONL algorithm increases the running time to $\Theta(n!n)$ thereby making the algorithm non-optimal. Hence, it is better to not use hashing with the PERMUTATIONL algorithm as shown in Figure 7.

(2) [PERMUTATIONR.] The algorithm uses the $\mathcal{R}_n$ permutation sequence. We use perfect hashing to compute SORTEDNEXT of a set of i elements. Using indexing (a perfect hashing technique) and consuming an extra space of $\Theta(n)$, these i elements can be sorted with $\Theta(n)$ comparisons. This implies that the total time required for all comparisons to generate permutations is $\Theta(n!n)$. Hence, the algorithm is non-optimal.

What if we use sorting for SORTEDNEXT? Using optimal comparison-sorting algorithm to sort items in the PERMUTATIONR algorithm increases the running time of the algorithm to $\Theta(n!n \log n)$. Hence, it is better to not use sorting with the PERMUTATIONR algorithm as shown in Figure 7.

Algorithm	Hashing	Sorting	Th.
PERMUTATIONL	$\Theta(n!n)$	$\Theta(n!)$	2.1
PERMUTATIONR	$\Theta(n!n)$	$\Theta(n!n \log n)$	2.2

FIGURE 7. Complexities of PERMUTATIONL and PERMUTATIONR algorithms. The complexity is affected by the average value of $\mathcal{L}_n$ and $\mathcal{R}_n$ sequences and the data structures and/or algorithms used to implement SORTEDNEXT.

5. CONCLUSION AND FUTURE WORK

We presented two simple, intuitive, and unified algorithmic frameworks — FRAMEWORKL and FRAMEWORKR, that was used to design 21 permutation generation algorithms, including the well-known algorithms of Wells, Langdon, Zaks, Tompkins, Lipski, and Heap. The two frameworks are based on two permutation sequences. We used the permutation frameworks to discover two new sorting-based permutation generation algorithms — PERMUTATIONL and PERMUTATIONR. We proved that the PERMUTATIONL algorithm is optimal.

Here are a few directions for future research:

1. Design the most efficient implementations of different permutation algorithms derivable from the two frameworks. (As an example, we strongly believe that the Johnson-Trotter's algorithm [10, 11] can be derived from FRAMEWORKR [1]. We plan to investigate if it is possible to derive this algorithm from FRAMEWORKR using multiple approaches such as naïve approach [9], Heap-like function (Algorithm 3a in [1]), inverse permutation [36], and loopless algorithms ([36, 12], Algorithm 3b in [1]).)
2. Are all permutation algorithms that use the sequence $\mathcal{L}_n$ (or $\mathcal{R}_n$) derivable from the two frameworks?
3. Are there permutation sequences other than $\mathcal{L}_n$ and $\mathcal{R}_n$?
4. Are there more permutation generation algorithms that can be designed using the frameworks?
5. Does there exist a unified permutation framework from which all permutation algorithms can be derived?
6. Are there patterns/sequences and algorithmic frameworks to design several other important combinatorial objects?

6. FUNDING

This work was supported in part by National Science Foundation grants [CCF-1162196, CCF-1439084, CNS-1553510].

7. DATA AVAILABILITY

No new input data were generated or analysed in support of this research.

Acknowledgements. We are grateful to reviewers for their invaluable suggestions. We also thank Abhiram Natarajan and Rama Badrinath for discussions.

REFERENCES

[1] Sedgewick, R. (1977) Permutation generation methods. *ACM Computing Surveys*, **9(2)**, 137–164.

[2] Williams, A. (2009) Loopless generation of multiset permutations using a constant number of variables by prefix shifts. *Proc. Symposium on Discrete Algorithms*, pp. 987–996.

[3] Wells, M. B. (1961) Generation of permutations by transposition. *Mathematics of Computation*, **15(74)**, 192–195.

[4] Heap, B. (1963) Permutations by interchanges. *The Computer Journal*, **6(3)**, 293–298.

[5] Ives, F. (1976) Permutation enumeration: four new permutation algorithms. *Communications of the ACM*, **19(2)**, 68–72.

[6] Fike, C. (1975) A permutation generation method. *The Computer Journal*, **18(1)**, 21–22.

[7] Rohl, J. S. (1976) Programming improvements to Fike's algorithm for generating permutations. *The Computer Journal*, **19(2)**, 156–159.

[8] Lipski Jr, W. (1979) More on permutation generation methods. *Computing*, **23(4)**, 357–365.

[9] Levitin, A. (2012) *Introduction to the Design and Analysis of Algorithms, 3/E*. Pearson.

[10] Johnson, S. M. (1963) Generation of permutations by adjacent transposition. *Mathematics of Computation*, **17(83)**, 282–285.

[11] Trotter, H. (1962) Algorithm 115: Perm. *Communications of the ACM*, **5(8)**, 434–435.

[12] Dershowitz, N. (1975) A simplified loop-free algorithm for generating permutations. *BIT Numerical Mathematics*, **15(2)**, 158–164.

[13] Zaks, S. (1984) A new algorithm for generation of permutations. *BIT Numerical Mathematics*, **24(2)**, 196–204.

[14] Rohl, J. S. (1978) Generating permutations by choosing. *The Computer Journal*, **21(4)**, 302–305.

[15] Langdon Jr, G. G. (1967) An algorithm for generating permutations. *Communications of the ACM*, **10(5)**, 298–299.

[16] Myrvold, W. and Ruskey, F. (2001) Ranking and unranking permutations in linear time. *Information Processing Letters*, **79(6)**, 281–284.

[17] Howell, J. R. (1962) Generation of permutations by addition. *Mathematics of Computation*, **16(78)**, 243–244.

[18] Arndt, J. (2010) *Matters Computational: Ideas, Algorithms, Source Code*. Springer-Verlag.

[19] Ord-Smith, R. (1970) Generation of permutation sequences: Part 1. *The Computer Journal*, **13(3)**, 152–155.

[20] Ord-Smith, R. (1971) Generation of permutation sequences: Part 2. *The Computer Journal*, **14(2)**, 136–139.

[21] Roy, M. K. (1978) Evaluation of permutation algorithms. *The Computer Journal*, **21(4)**, 296–301.

[22] Gupta, P. and Bhattacherjee, G. (1983) Parallel generation of permutations. *The Computer Journal*, **26(2)**, 97–105.

[23] Reif, J. H. (1985) An optimal parallel algorithm for integer sorting. *Foundations of Computer Science*, pp. 496–504.

[24] Chen, G.-H. and Chern, M.-S. (1986) Parallel generation of permutations and combinations. *BIT Numerical Mathematics*, **26(3)**, 277–283.

[25] Akl, S. G. (1987) Adaptive and optimal parallel algorithms for enumerating permutations and combinations. *The Computer Journal*, **30(5)**, 433–436.

[26] Anderson, R. J. (1990) Parallel algorithms for generating random permutations on a shared memory machine. *Proc. Symposium on Parallel Algorithms and Architectures*, pp. 95–102. ACM.

[27] Korsh, J. and Lipschutz, S. (1997) Generating multiset permutations in constant time. *Journal of Algorithms*, **25(2)**, 321–335.

[28] Korsh, J. F. and LaFollette, P. S. (2004) Loopless array generation of multiset permutations. *The Computer Journal*, **47(5)**, 612–621.

[29] Chowdhury, R., Ganapathi, P., Tschudi, S., Tithi, J. J., Bachmeier, C., Leiserson, C. E., Solar-Lezama, A., Kuszmaul, B. C., and Tang, Y. (2017) Autogen: Automatic Discovery of Efficient Recursive Divide-&-Conquer Algorithms for Solving Dynamic Programming Problems. *ACM Transactions on Parallel Computing*, **4**, 1–30.

[30] Ganapathi, P. (2016) Automatic Discovery of Efficient Divide-&-Conquer Algorithms for Dynamic Programming Problems. PhD thesis Department of Computer Science, State University of New York at Stony Brook.

[31] Knuth, D. E. (2011) *The Art of Computer Programming Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley.

[32] Hartung, E., Hoang, H. P., Mütze, T., and Williams, A. (2020) Combinatorial generation via permutation languages. *Proc. Symposium on Discrete Algorithms*, pp. 1214–1225.

[33] (2020). OEIS Sequence A055881. `https://oeis.org/A055881`. [Online; accessed 24-February-2020].

[34] Rohl, J. S. (1991) Ord Smith's pseudo-lexicographical permutation procedure is the Tompkins-Paige algorithm. *The Computer Journal*, **34(6)**, 569–570.

[35] Ruskey, F. (2003) Combinatorial generation. *Working draft. University of Victoria*, **11**, 1–311.

[36] Ehrlich, G. (1973) Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *Journal of the ACM*, **20(3)**, 500–513.