Premtuatoins and PAttERns*

Rezaul Chowdhury and Pramod Ganapathi

Department of Computer Science, Stony Brook University, New York, USA.

Abstract

We describe two patterns that are used to generate permutations of a distinct set of elements. We use the patterns to discover two permutation generation algorithms that are based on sorting. Finally, we present two simple and general frameworks that uses the patterns to generate 21 permutation algorithms including the well-known algorithms of Wells, Langdon, Zaks, Tompkins, Lipski, and Heap. Our frameworks are simple and intuitive.

1998 ACM Subject Classification G.2.1 Combinatorics

Keywords and phrases permutation generation, permutation, pattern, framework, sort

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Science has often been considered as the study of patterns in nature. In the most general sense, a *pattern* is any kind of regularity that can be explained by a scientific theory [9]. They are the generalizations of different but related instances. Patterns are everywhere. Mathematical formulas are the algebraic patterns that show the relation between certain numbers; mathematical functions are the geometric patterns that describe the relation between independent and dependent variables; algorithms are the step-by-step procedure patterns to solve mostly computer science problems; predictions of the future (e.g.: weather forecasting, planetary motion, etc) are based on the generalized patterns of observed events in some prediction models (such as statistics, machine learning, probability theory, and theories of physics); medical diagnosis uses disease / illness patterns to determine the disease or illness of a person based on the person's symptoms; clothes / buildings with patterns are aesthetically beautiful; storylines and philosophies of time-loop patterns have been excellently depicted in mind-blowing movies such as *Predestination* (2014), *Triangle* (2009), and *The Infinite Man* (2014); and there exists a theory that an ith dimensional universe is present on the event horizon of an $(i+1)$th dimensional universe [24].

In this paper, we study two patterns of permutation generation algorithms and use them to discover two new permutation algorithms based on sorting. We also use the same two patterns to construct two simple frameworks, which in turn can be used to develop several permutation algorithms.

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 cute little problem.

* The word *premtuatoins* is a permutation of the word permutations. The word *PAttERns* has a pattern of alternating two capital and two small letters.

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

There are many practical applications that require permutation generation. A few potential applications include: (*a*) *Software testing* [30]: to test the performance of an operating system by executing all permutations of a set of tasks; (*b*) *Communication networks, cryptography, and network security* [33]: in error detection and correction codes that enable reliable delivery of digital data over unreliable communication channels; (*c*) *Randomized algorithms*: to generate random permutations in coding theory and simulations; and (*d*) *Operations research*: in problems such as traveling salesman (TSP).

Permutation generation algorithms can be classified into different categories based on how they generate the next permutation. Permutations of a set of n distinct elements are generated based on: (*a*) *Swaps* [32, 11, 13, 7, 26, 20]: where two elements are interchanged; (*b*) *Adjacent swaps* [19, 14, 31, 6]: where two adjacent elements are interchanged; (*c*) *Reversals* [34]: where a certain prefix or suffix of a permutation is reversed; (*d*) *Counters* [27]: where counts of the elements are decremented and incremented; (*e*) *Rotations* [18, 33]: where a certain prefix or suffix of a permutation is left- or right-rotated; (*f*) *Unranking* [34, 21]: where a number from 1 to $n!$ is mapped on to a permutation; and (*g*) *Additions* [12]: 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 [4, 30, 22, 23, 29]. In the recent three decades, the focus has been towards developing parallel algorithms to generate permutations [10, 25, 5, 2, 3] as well as coming up with permutation algorithms for a multiset [16, 17, 33].

Sedgewick [30], in his survey paper, analyzes different permutation algorithms and identifies two permutation patterns. He observes that the control structure of many permutation algorithms are similar ([30], 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 patterns to discover permutation algorithms. Lipski [20] gives the first explicit scheme to produce a class of permutation algorithms based on swaps / interchanges. He gives 16 different algorithms based on swaps, that includes well-known algorithms of Heap [11] and Wells [32]. Knuth [15] gives an explicit framework / generic permutation generator ([15], 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 $i \in [2, n]$.

In our paper, we construct frameworks from the permutation patterns 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 way of constructing the frameworks is different. Hence, our frameworks are simpler and intuitive. Also, to the best of our knowledge, our second framework has not been studied in literature.

The time complexity of individual permutation algorithms does not matter when aiming for a unified theory to represent permutation algorithms. The most important characteristic is the flexibility and generality of the framework or model which helps in the deeper understanding of the relation (or similarities) between various seemingly unrelated permutation algorithms. This knowledge aids in better algorithm design.

Our contributions. Our results can be divided into two major categories:

(1) [Permutation algorithms (Section 3).] We use the existing permutation patterns to discover two permutation generation algorithms that are based on sorting and prove their correctness. One of the two algorithms is asymptotically fastest.

(2) [Permutation frameworks (Section 4).] We present two simple and general frameworks using the patterns that enable us to generate 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 method known).

Organization of the paper. The paper is organized as follows. In Section 2, we describe two patterns used to generate permutations. In Section 3, we present two new algorithms that use patterns and sorting to generate permutations of a set of distinct elements. In Section 4, we give a general framework that uses patterns to produce a class of permutation algorithms.

2 Patterns

In this section, we describe two permutation patterns, present algorithms to generate them, prove their correctness, and describe their properties.

A permutation pattern, or simply a pattern, 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 patterns called left and right patterns, denoted by $\mathcal{L}_n$ and $\mathcal{R}_n$, respectively, are as follows:

$$\mathcal{L}_n = \begin{cases} 2 & \text{if } n = 2, \\ \mathcal{L}_{n-1}(n\mathcal{L}_{n-1})^{n-1} & \text{if } n > 2, \end{cases} \quad \mathcal{R}_n = \begin{cases} 2 & \text{if } n = 2, \\ n^{n-1}(\mathcal{R}_{n-1}[i]n^{n-1})^{(n-1)!-1} & \text{if } n > 2, \end{cases}$$

where i varies from 1 to $(n-1)! - 1$ and $\mathcal{R}_{n-1}[i]$ denotes the i^{th} character of $\mathcal{R}_{n-1}$. The first few values of $\mathcal{L}_n$ are $\mathcal{L}_2 = 2$, $\mathcal{L}_3 = 23232$, and $\mathcal{L}_4 = 23232423232423232423232$. Similarly, the first few values of $\mathcal{R}_n$ are $\mathcal{R}_2 = 2$, $\mathcal{R}_3 = 33233$, and $\mathcal{R}_4 = 44434443444244434443444$. The patterns $\mathcal{L}_n$ and $\mathcal{R}_n$ are of size $n! - 1$.

The two patterns have been previously described in [6, 30, 34], and several other places [4, 1]. In this section, we give simple algorithms to generate the patterns and in further sections we use these patterns to discover new permutation algorithms.

2.1 Algorithms

The recursive and iterative algorithms to generate the two patterns are given in Table 1. Algorithm IPatternL and Lemma 1 are due to Shmuel Zaks [34] and have been incorporated here for completeness. The correctness proofs for the other algorithms are not given due to space constraints.

2.2 Properties

We give bounds for the average value of the integers in the two patterns. Lemma 1 is due to Zaks.

▶ **Lemma 1** (Zaks [34]: $\mathcal{L}_n$ average)**.** *The average value of $\mathcal{L}_n$ is upper bounded by the Euler's number, e.*

▶ **Lemma 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 $x_n = ((n-1)! - 1)/(n! - 1)$.

```
PatternL(k)
Input: k; Output: L_n
Invoke: PatternL(n)
1. if k = 2 then print k
2. PatternL(k − 1)
3. for i ← k to 2 do
4.    print k
5.    PatternL(k − 1)
```

```
PatternR(k)
Input: k; Output: R_n
Invoke: PatternR(2)
1. if k = n then
2.    for i ← k to 2 do
3.       print k
4.    PatternR(k + 1)
5. for i ← k to 2 do
6.    print k
7.    PatternR(k + 1)
```

```
IPatternL()
Output: 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.    print 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
```

```
IPatternR(k)
Output: 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.       print k
8.       k ← n
9.    else
10.      for i ← k to 2 do
11.         print k
12.      k ← next[n]
13.      next[n] ← n − 1
14.      count[k] ← count[k] + 1
15.      if count[k] = k−1 then
16.         count[k] ← 0
17.         next[k + 1] ← next[k]
18.         next[k] ← k − 1
```

Table 1 Recursive & iterative algorithms to generate the two permutation patterns: $\mathcal{L}_n$, $\mathcal{R}_n$.

We use mathematical induction to prove the lemma. 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. ◀

3 Permutations

In this section, we present two permutation generation algorithms based on the patterns described in the previous section and prove their correctness.

A permutation $\mathcal{P} = [p_1, p_2, \ldots, p_n]$ 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 integers from 1 to n. The algorithms we are going to describe in subsequent sections uses a global variable $\mathcal{P}$ to store permutations, which is initialized to $[1, 2, \ldots, n]$. Whenever a new permutation is generated, it is stored in $\mathcal{P}$ and then printed.

3.1 Algorithms

We present two permutation generation algorithms called PermutationL and PermutationR. The two algorithms are based on left and right patterns: $\mathcal{L}_n$ and $\mathcal{R}_n$. The recursive algorithms for PermutationL and PermutationR are given in Table 2. 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.

Algorithm PermutationL is derived from PatternL by replacing the line "**print** k" with the two lines "Func(k); **print** $\mathcal{P}$". The algorithm PermutationL basically generates the left pattern $\mathcal{L}_n$, which is a sequence of $n! - 1$ integers and sends each of these integers, k, as an argument to a 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 $n!$ unique permutations of $[1, 2, \ldots, n]$. In a similar way, PermutationR, too, generates all the $n!$ unique permutations of $[1, 2, \ldots, n]$ using the right pattern $\mathcal{R}_n$.

PermutationL(k)

Input: k, global $\mathcal{P} = [1, 2, \ldots, n]$
Output: Permutations of $\mathcal{P}$
Invoke: PermutationL(n)

```
1. if k = 2 then
2.    Func(k)
3.    print P
4. PermutationL(k − 1)
5. for i ← k to 2 do
6.    Func(k)
7.    print P
8.    PermutationL(k − 1)
```

PermutationR(k)

Input: k, global $\mathcal{P} = [1, 2, \ldots, n]$
Output: Permutations of $\mathcal{P}$
Invoke: PermutationR(2)

```
1. if k = n then
2.    for i ← k to 2 do
3.       Func(k)
4.       print P
5. PermutationR(k + 1)
6. for i ← k to 2 do
7.    Func(k)
8.    print P
9.    PermutationR(k + 1)
```

Func(k)

Input: k, global $\mathcal{P}$
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])
```

Table 2 Recursive permutation algorithms PermutationL and PermutationR to generate permutations of $\mathcal{P} = [1, 2, \ldots, n]$.

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 $n - k$ of $\mathcal{P}$. The technique is called SortedNext, which is defined as follows.

▶ **Definition 3** (SortedNext)**.** Let $[a_1, a_2, \ldots, a_m]$ be a tuple of m elements from a totally ordered universe, and let $S = [s_1, s_2, \ldots, s_m]$ be those m elements in sorted order. Also, let Rank(a_i), where $i \in [1, m]$, represent the position of an element a_i in S i.e., Rank(a_i) $= j \in [1, m]$ such that $a_i = s_j$. Then, SortedNext for a list of elements is defined as SortedNext($[a_1, a_2, \ldots, a_m]$) $= [x_1, x_2, \ldots, x_m]$, where $x_i = s_{1+(\text{Rank}(a_i) \pmod m)}$ for $i \in [1, m]$.

The recursion trees for generating permutations, also called permutation trees are illustrated in Figure 1. Let Func(k) in Table 2 be denoted by $\mathcal{F}_k$. Then, the permutations of $[1, 2, 3]$ as generated by PermutationL and PermutationR, respectively, are:

$$\text{PermutationL}: \quad [1,2,3] \xrightarrow{\mathcal{F}_2} [1,3,2] \xrightarrow{\mathcal{F}_3} [2,1,3] \xrightarrow{\mathcal{F}_2} [2,3,1] \xrightarrow{\mathcal{F}_3} [3,1,2] \xrightarrow{\mathcal{F}_2} [3,2,1]$$

$$\text{PermutationR}: \quad [1,2,3] \xrightarrow{\mathcal{F}_3} [2,3,1] \xrightarrow{\mathcal{F}_3} [3,1,2] \xrightarrow{\mathcal{F}_2} [3,2,1] \xrightarrow{\mathcal{F}_3} [1,3,2] \xrightarrow{\mathcal{F}_3} [2,1,3]$$

The PermutationR algorithm is developed by Anil Bhandary and the second author [8].

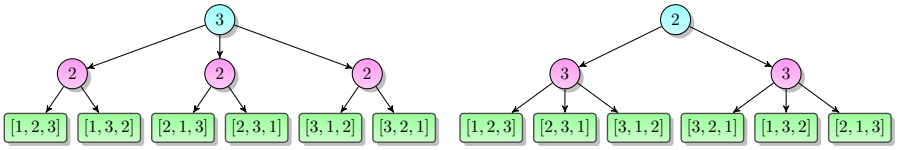

Figure 1 Permutation trees for PERMUTATIONL and PERMUTATIONR $\mathcal{P} = [1, 2, 3]$.

3.2 Proofs of correctness

We give correctness proofs for the two permutation generation algorithms. In our proofs we denote FUNC(k) by $\mathcal{F}_k$.

▶ **Theorem 4** (PERMUTATIONL correctness). PERMUTATIONL *generates all* $n!$ *unique permutations of* $[1, 2, \ldots, n]$.

Proof. We use mathematical 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 5** (PERMUTATIONR correctness). PERMUTATIONR *generates all* $n!$ *unique permutations of* $[1, 2, \ldots, n]$.

Proof. We use mathematical 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 $\mathcal{R}_{n+1} = (n + 1)^n(\mathcal{R}_n[i](n + 1)^n)^{n!-1}$, where $\mathcal{R}_n[i]$ is the i^{th} integer 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, 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. Then $\mathcal{B}_i$ 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 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. ◀

3.3 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 $\Theta(n)$ words. The time complexity to print the permutations is $\Omega(n!n)$ for any permutation algorithm. The time complexities to simply compute all the permutations are as follows:

(1) [PermutationL.] The algorithm uses $\mathcal{L}_n$ and in $\mathcal{L}_n$ an element i occurs $c(n,i) = n!(i-1)/i!$ times. We use mergesort to compute SortedNext of a set of i elements. The total time required for all comparisons to generate permutations is $\Theta\left(\sum_{i=2}^{n} c(n,i) \cdot i \log i\right) = \Theta\left(n! \sum_{i=2}^{n} (\log i)/(i-2)!\right) = \Theta\left(n! + n! \sum_{i=4}^{n} (\log i)/(i-2)!\right) = \Theta\left(n! + n! \sum_{i=4}^{n} 1/(i-3)!\right) = \Theta(n!)$. As the amortized cost of computing the next permutation is $\mathcal{O}(1)$, the algorithm is one of the *theoretically fastest* permutation algorithms.

(2) [PermutationR.] The algorithm uses $\mathcal{R}_n$. We use perfect hashing to compute SortedNext of a set of i elements. Using indexing, a perfect hashing technique, with an extra space of $\Theta(n)$ words, these i elements can be sorted in $\Theta(n)$ comparisons. Hence, the total time required for all comparisons to generate permutations is $\Theta(n!n)$.

4 Frameworks

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

A permutation framework, or simply a framework, is a generic structure to obtain a class of permutation generation algorithms. Table 3 shows two frameworks with an empty function Func. The frameworks FrameworkL and FrameworkR basically generate the patterns $\mathcal{L}_n$ (PatternL) and $\mathcal{R}_n$ (PatternR) and are almost similar to PermutationL and PermutationR, respectively. Both the frameworks invoke Func with 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 $\mathcal{P}$.

Algorithms	Func	FL	FR	Ref.
Heap	Swap	✓	–	[30, 20, 11]
Wells	Swap	✓	–	[30, 20, 32]
Lipski	Swap	✓	–	[20]
Zaks	Reverse	✓	–	[34]
Langdon	Rotate	–	✓	[30, 18]
Tompkins	Rotate	✓	–	[30, 28][*]
Proposed	SortedNext	✓	✓	[*]

Table 4 Func functions for 21 permutation algorithms for two frameworks: FrameworkL and FrameworkR, denoted by FL and FR, respectively. * represents this paper.

Table 3 gives 6 definitions of the Func function that lead to 7 different permutation algorithms. By defining the function Func differently i.e., by giving different meanings to the integer k of the two patterns and the loop parameter i, we end up with 21 different permutation generation algorithms.

```
FrameworkL(k)
Input: k, global P = [1, 2, ..., n]
Output: Permutations of P
Invoke: FrameworkL(n)
1. if k = 2 then
2.    Func(k, k)
3.    print P
4. FrameworkL(k − 1)
5. for i ← k to 2 do
6.    Func(k, i)
7.    print P
8.    FrameworkL(k − 1)
```

```
FrameworkR(k)
Input: k, global P = [1, 2, ..., n]
Output: Permutations of P
Invoke: FrameworkR(2)
1. if k = n then
2.    for i ← k to 2 do
3.       Func(k, i)
4.       print P
5. FrameworkR(k + 1)
6. for i ← k to 2 do
7.    Func(k, i)
8.    print P
9.    FrameworkR(k + 1)
```

```
Func(k, i)
Input: k, i, global P
Output: Next permutation of P
1. { Any good working function }
```

```
Func(k, i)
1. { Proposed function for FL and FR }
2. SortedNext([p_{n−k+1}, p_{n−k+2}, ..., p_n])
```

```
Func(k, i)
1. { Zaks' function for FL }
2. Reverse([p_{n−k+1}, p_{n−k+2}, ..., p_n])
```

```
Func(k, i)
1. { Langdon's function for FR }
2. for j ← n to k do
3.    Rotate([p_{n−j+1}, p_{n−j+2}, ..., p_n])
```

```
Func(k, i)
1. { Tompkins's function for FL }
2. for j ← 2 to k do
3.    Rotate([p_{n−j+1}, p_{n−j+2}, ..., p_n])
```

```
Func(k, i)
1. { Heap's function for FL }
2. j ← k − i + 1
3. if k % 2 = 0 then Swap(p_k, p_j)
4. else Swap(p_k, p_1)
```

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

Table 3 Left column: Two general frameworks to generate permutations of $\mathcal{P} = [1, 2, \ldots, n]$ using the two patterns $\mathcal{L}_n$ and $\mathcal{R}_n$. Right column: Six definitions of the Func function used to produce seven permutation generation algorithms. FL and FR means FrameworkL and FrameworkR, respectively.

The proposed function in Table 3 was used in PermutationL and PermutationR algorithms and was introduced in Section 3. The function works for both FrameworkL and FrameworkR and applies the SortedNext to the last k elements of the current permutation in $\mathcal{P}$ to get a new permutation. Zaks' function for FrameworkL uses Reverse to reverse the last k elements of the current permutation. Langdon's function for FrameworkR applies a sequence of Rotates to rotate the last j ($j \leftarrow n$ to k) elements of the current permutation. Similarly, Tompkins-Paige's function for FrameworkL applies a sequence of Rotate's to rotate the last j ($j \leftarrow 2$ to k) elements of the current permutation. All the above function definitions do not make use of i and 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.

Heap's and Wells' functions for FrameworkL use the Swap procedure to interchange two elements of the current permutation, one of them being the kth element. Similar to Heap and Wells functions, we can produce 14 more permutation algorithms developed by Lipski [20], which use the Swap procedure. Several permutation generation algorithms, the frameworks used to produce them, and the strategies of their Func functions are summarized in Table 4.

Acknowledgements. Chowdhury and Ganapathi were supported in part by NSF grants CCF-1162196 and CCF-1439084. We thank Abhiram Natarajan, Rama Badrinath, and anonymous referees for useful suggestions.

References

1 OEIS Sequence A055881. `https://oeis.org/A055881`.

2 Selim G Akl. Adaptive and optimal parallel algorithms for enumerating permutations and combinations. *The Computer Journal*, pages 30(5):433–436, 1987.

3 Richard J Anderson. Parallel algorithms for generating random permutations on a shared memory machine. In *Symposium on Parallel Algorithms and Architectures*, pages 95–102. ACM, 1990.

4 Jorg Arndt. *Matters Computational: Ideas, Algorithms, Source Code*. Springer-Verlag New York, Inc., 2010.

5 Gen-Huey Chen and Maw-Sheng Chern. Parallel generation of permutations and combinations. *BIT Numerical Mathematics*, pages 26(3):277–283, 1986.

6 Nachum Dershowitz. A simplified loop-free algorithm for generating permutations. *BIT Numerical Mathematics*, pages 15(2):158–164, 1975.

7 CT Fike. A permutation generation method. *The Computer Journal*, pages 18(1):21–22, 1975.

8 Pramod Ganapathi and Anil Bhandary. An algorithm to generate permutations based on rotations. *Unpublished*, 2008.

9 Ulf Grenander and Michael I Miller. *Pattern theory: from representation to inference*, volume 1. 2007.

10 Phalguni Gupta and GP Bhattacherjee. Parallel generation of permutations. *The Computer Journal*, pages 26(2):97–105, 1983.

11 BR Heap. Permutations by interchanges. *The Computer Journal*, pages 6(3):293–298, 1963.

12 John R Howell. Generation of permutations by addition. *Mathematics of Computation*, pages 16(78):243–244, 1962.

13 FM Ives. Permutation enumeration: four new permutation algorithms. *CACM*, pages 19(2):68–72, 1976.

14 S M Johnson. Generation of permutations by adjacent transposition. *Mathematics of Computation*, pages 17(83):282–285, 1963.

15 Donald E. Knuth. *The Art of Computer Programming Volume 4A: Combinatorial Algorithms, Part 1.* Addison-Wesley Publishing Company, 2011.

16 James Korsh and Seymour Lipschutz. Generating multiset permutations in constant time. *Journal of Algorithms*, pages 25(2):321–335, 1997.

17 James F Korsh and Paul S LaFollette. Loopless array generation of multiset permutations. *The Computer Journal*, pages 47(5):612–621, 2004.

18 Glen G Langdon Jr. An algorithm for generating permutations. *CACM*, pages 10(5):298–299, 1967.

19 Anany Levitin. *Introduction to the Design and Analysis of Algorithms, 3/E.* Pearson, 2012.

20 W Lipski Jr. More on permutation generation methods. *Computing*, pages 23(4):357–365, 1979.

21 Wendy Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, pages 79(6):281–284, 2001.

22 RJ Ord-Smith. Generation of permutation sequences: Part 1. *The Computer Journal*, pages 13(3):152–155, 1970.

23 RJ Ord-Smith. Generation of permutation sequences: Part 2. *The Computer Journal*, pages 14(2):136–139, 1971.

24 Razieh Pourhasan, Niayesh Afshordi, and Robert B Mann. Out of the white hole: a holographic origin for the big bang. *Journal of Cosmology and Astroparticle Physics*, pages 04:1–12, 2014.

25 John H Reif. An optimal parallel algorithm for integer sorting. In *Foundations of Computer Science*, pages 496–504, 1985.

26 Jeffrey S. Rohl. Programming improvements to fike's algorithm for generating permutations. *The Computer Journal*, pages 19(2):156–159, 1976.

27 Jeffrey S. Rohl. Generating permutations by choosing. *The Computer Journal*, pages 21(4):302–305, 1978.

28 Jeffrey S. Rohl. Ord smith's pseudo-lexicographical permutation procedure is the tompkins-paige algorithm. *The Computer Journal*, pages 34(6):569–570, 1991.

29 Mohit Kumar Roy. Evaluation of permutation algorithms. *The Computer Journal*, pages 21(4):296–301, 1978.

30 Robert Sedgewick. Permutation generation methods. *ACM Computing Surveys*, pages 9(2):137–164, 1977.

31 HF Trotter. Algorithm 115: Perm. *CACM*, pages 5(8):434–435, 1962.

32 Mark B Wells. Generation of permutations by transposition. *Mathematics of Computation*, pages 15(74):192–195, 1961.

33 Aaron Williams. Loopless generation of multiset permutations using a constant number of variables by prefix shifts. In *Symposium on Discrete Algorithms*, pages 987–996, 2009.

34 Shmuel Zaks. A new algorithm for generation of permutations. *BIT Numerical Mathematics*, pages 24(2):196–204, 1984.

A Proofs of the algorithms that generate the patterns

▶ **Theorem 6** (PatternL correctness). *Algorithm* PatternL *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 PatternL$(n+1)$. We see that the function PatternL(n) is called $n+1$ times and between every two successive calls the integer $n+1$ is generated once. Clearly, the pattern 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 7** (PatternR correctness). *Algorithm* PatternR *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 PatternR(2), which in turn calls PatternR(3) and the process continues. The function PatternR(n) calls PatternR$(n+1)$ n times. In every call of PatternR$(n+1)$, 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 PatternR$(n+1)$. Clearly, the pattern generation follows the rule of $\mathcal{R}_{n+1} = (n+1)^n(\mathcal{R}_n[i](n+1)^n)^{n!-1}$, where $\mathcal{R}_n[i]$ is an integer of $\mathcal{R}_n$. Thus, $S(n+1)$ is true. ◀

▶ **Theorem 8** (Zaks [34]: IPatternL correctness). *Algorithm* IPatternL *generates* $\mathcal{L}_n$.

Proof. (simplified from Zaks [34].) 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 IPatternL and Table 5.

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 $count[n+1] = 2$. 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 9** (IPatternR correctness). *Algorithm* IPatternR *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 IPatternR and Table 6.

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 $(n+1)^n(\mathcal{R}_n[i](n+1)^n)^{n!-1}$ and due to our assumption that $S(n)$ is true the configuration becomes $k = 1$ from line 12, $next[n+1] = n$ from line 13, and $count[1] = 1$ from line 14. As $k = 1$, the control exits from the while loop and the algorithm terminates. Thus, $S(n+1)$ is true. ◀

B More functions of FrameworkL and FrameworkR

Table 7 summarizes the sixteen definitions of Func function defined by Lipski [20] that make use of the Swap procedure.

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

Table 5 Steps in the proof of Theorem 8.

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

Table 6 Steps in the proof of Theorem 9.

FUNC(k,i) { Lipski algorithm 1 }

1. $j \leftarrow k-i+1$
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 algorithm 2 }

1. $j \leftarrow k-i+1$
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 algorithm 3 }

1. $j \leftarrow k-i+1$
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 algorithm 4 }

1. $j \leftarrow k-i+1$
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 algorithm 5 }

1. $j \leftarrow k-i+1$
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 algorithm 6 }

1. $j \leftarrow k-i+1$
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 algorithm 7 }

1. $j \leftarrow k-i+1$
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 algorithm 8 }

1. $j \leftarrow k-i+1$
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 algorithm 9 }

1. $j \leftarrow k-i+1$
2. **if** $k \,\%\, 2 = 0$ **then** SWAP(p_k, p_j)
3. **else** SWAP(p_k, p_1)

FUNC(k,i) { Lipski algorithm 10 }

1. $j \leftarrow k-i+1$
2. **if** $k \,\%\, 2 = 0$ **then** SWAP(p_k, p_j)
3. **else** SWAP(p_k, p_{k-2})

FUNC(k,i) { Lipski algorithm 11 }

1. $j \leftarrow k-i+1$
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 algorithm 12 }

1. $j \leftarrow k-i+1$
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 algorithm 13 }

1. $j \leftarrow k-i+1$
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 algorithm 14 }

1. $j \leftarrow k-i+1$
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 algorithm 15 }

1. $j \leftarrow k-i+1$
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 algorithm 16 }

1. $j \leftarrow k-i+1$
2. **if** $k \,\%\, 2 = 0$ **then** SWAP(p_k, p_{k-j})
3. **else** SWAP(p_k, p_1)

Table 7 16 definitions of the FUNC functions as defined by Lipski [20] for FRAMEWORKL.

$n=2;\ \mathcal{P}=[1,2]$				$n=3;\ \mathcal{P}=[1,2,3]$				$n=4;\ \mathcal{P}=[1,2,3,4]$			
$\mathcal{L}_2$	PL	$\mathcal{R}_2$	PR	$\mathcal{L}_3$	PL	$\mathcal{R}_3$	PR	$\mathcal{L}_4$	PL	$\mathcal{R}_4$	PR
	$[1,2]$		$[1,2]$		$[1,2,3]$		$[1,2,3]$		$[1,2,3,4]$		$[1,2,3,4]$
2	$[2,1]$	2	$[2,1]$	2	$[1,3,2]$	3	$[2,3,1]$	2	$[1,2,4,3]$	4	$[2,3,4,1]$
				3	$[2,1,3]$	3	$[3,1,2]$	3	$[1,3,2,4]$	4	$[3,4,1,2]$
				2	$[2,3,1]$	2	$[3,2,1]$	2	$[1,3,4,2]$	4	$[4,1,2,3]$
				3	$[3,1,2]$	3	$[1,3,2]$	3	$[1,4,2,3]$	3	$[4,2,3,1]$
				2	$[3,2,1]$	3	$[2,1,3]$	2	$[1,4,3,2]$	4	$[1,3,4,2]$
								4	$[2,1,4,3]$	4	$[2,4,1,3]$
								2	$[2,1,3,4]$	4	$[3,1,2,4]$
								3	$[2,3,4,1]$	3	$[3,2,4,1]$
								2	$[2,3,1,4]$	4	$[4,3,1,2]$
								3	$[2,4,3,1]$	4	$[1,4,2,3]$
								2	$[2,4,1,3]$	4	$[2,1,3,4]$
								4	$[3,1,2,4]$	2	$[2,1,4,3]$
								2	$[3,1,4,2]$	4	$[3,2,1,4]$
								3	$[3,2,1,4]$	4	$[4,3,2,1]$
								2	$[3,2,4,1]$	4	$[1,4,3,2]$
								3	$[3,4,1,2]$	3	$[1,2,4,3]$
								2	$[3,4,2,1]$	4	$[2,3,1,4]$
								4	$[4,1,3,2]$	4	$[3,4,2,1]$
								2	$[4,1,2,3]$	4	$[4,1,3,2]$
								3	$[4,2,3,1]$	3	$[4,2,1,3]$
								2	$[4,2,1,3]$	4	$[1,3,2,4]$
								3	$[4,3,2,1]$	4	$[2,4,3,1]$
								2	$[4,3,1,2]$	4	$[3,1,4,2]$

Table 8 Permutations of $\mathcal{P} = [1, 2, \ldots, n]$ for $n = 2, 3$, and 4 using PermutationL and PermutationR algorithms, denoted by PL and PR, respectively.

PL	PR	Heap	Wells	Zaks	Langdon	Tompkins Paige
$[1,2,3,4]$	$[1,2,3,4]$	$[1,2,3,4]$	$[1,2,3,4]$	$[1,2,3,4]$	$[1,2,3,4]$	$[1,2,3,4]$
$[1,2,4,3]$	$[2,3,4,1]$	$[2,1,3,4]$	$[2,1,3,4]$	$[1,2,4,3]$	$[2,3,4,1]$	$[1,2,4,3]$
$[1,3,2,4]$	$[3,4,1,2]$	$[3,1,2,4]$	$[2,3,1,4]$	$[1,3,4,2]$	$[3,4,1,2]$	$[1,3,4,2]$
$[1,3,4,2]$	$[4,1,2,3]$	$[1,3,2,4]$	$[3,2,1,4]$	$[1,3,2,4]$	$[4,1,2,3]$	$[1,3,2,4]$
$[1,4,2,3]$	$[4,2,3,1]$	$[2,3,1,4]$	$[3,1,2,4]$	$[1,4,2,3]$	$[1,3,4,2]$	$[1,4,2,3]$
$[1,4,3,2]$	$[1,3,4,2]$	$[3,2,1,4]$	$[1,3,2,4]$	$[1,4,3,2]$	$[3,4,2,1]$	$[1,4,3,2]$
$[2,1,4,3]$	$[2,4,1,3]$	$[4,2,1,3]$	$[1,3,4,2]$	$[2,3,4,1]$	$[4,2,1,3]$	$[2,3,4,1]$
$[2,1,3,4]$	$[3,1,2,4]$	$[2,4,1,3]$	$[3,1,4,2]$	$[2,3,1,4]$	$[2,1,3,4]$	$[2,3,1,4]$
$[2,3,4,1]$	$[3,2,4,1]$	$[1,4,2,3]$	$[3,4,1,2]$	$[2,4,1,3]$	$[1,4,2,3]$	$[2,4,1,3]$
$[2,3,1,4]$	$[4,3,1,2]$	$[4,1,2,3]$	$[4,3,1,2]$	$[2,4,3,1]$	$[4,2,3,1]$	$[2,4,3,1]$
$[2,4,3,1]$	$[1,4,2,3]$	$[2,1,4,3]$	$[4,1,3,2]$	$[2,1,3,4]$	$[2,3,1,4]$	$[2,1,3,4]$
$[2,4,1,3]$	$[2,1,3,4]$	$[1,2,4,3]$	$[1,4,3,2]$	$[2,1,4,3]$	$[3,1,4,2]$	$[2,1,4,3]$
$[3,1,2,4]$	$[2,1,4,3]$	$[1,3,4,2]$	$[1,4,2,3]$	$[3,4,1,2]$	$[1,2,4,3]$	$[3,4,1,2]$
$[3,1,4,2]$	$[3,2,1,4]$	$[3,1,4,2]$	$[4,1,2,3]$	$[3,4,2,1]$	$[2,4,3,1]$	$[3,4,2,1]$
$[3,2,1,4]$	$[4,3,2,1]$	$[4,1,3,2]$	$[4,2,1,3]$	$[3,1,2,4]$	$[4,3,1,2]$	$[3,1,2,4]$
$[3,2,4,1]$	$[1,4,3,2]$	$[1,4,3,2]$	$[2,4,1,3]$	$[3,1,4,2]$	$[3,1,2,4]$	$[3,1,4,2]$
$[3,4,1,2]$	$[1,2,4,3]$	$[3,4,1,2]$	$[2,1,4,3]$	$[3,2,4,1]$	$[1,4,3,2]$	$[3,2,4,1]$
$[3,4,2,1]$	$[2,3,1,4]$	$[4,3,1,2]$	$[1,2,4,3]$	$[3,2,1,4]$	$[4,3,2,1]$	$[3,2,1,4]$
$[4,1,3,2]$	$[3,4,2,1]$	$[4,3,2,1]$	$[3,2,4,1]$	$[4,1,2,3]$	$[3,2,1,4]$	$[4,1,2,3]$
$[4,1,2,3]$	$[4,1,3,2]$	$[3,4,2,1]$	$[2,3,4,1]$	$[4,1,3,2]$	$[2,1,4,3]$	$[4,1,3,2]$
$[4,2,3,1]$	$[4,2,1,3]$	$[2,4,3,1]$	$[2,4,3,1]$	$[4,2,3,1]$	$[1,3,2,4]$	$[4,2,3,1]$
$[4,2,1,3]$	$[1,3,2,4]$	$[4,2,3,1]$	$[4,2,3,1]$	$[4,2,1,3]$	$[3,2,4,1]$	$[4,2,1,3]$
$[4,3,2,1]$	$[2,4,3,1]$	$[3,2,4,1]$	$[4,3,2,1]$	$[4,3,1,2]$	$[2,4,1,3]$	$[4,3,1,2]$
$[4,3,1,2]$	$[3,1,4,2]$	$[2,3,4,1]$	$[3,4,2,1]$	$[4,3,2,1]$	$[4,1,3,2]$	$[4,3,2,1]$

Table 9 Permutations of $\mathcal{P} = [1, 2, 3, 4]$ using different algorithms. The PermutationL and PermutationR algorithms are denoted by PL and PR, respectively.