全排列生成复杂度分析与基于对称性的加速算法

郭嘉懿, 杜超群 & 吴玉婷
清华大学自动化系
{guo-jy20,dcq20,wyt20}@mails.tsinghua.edu.cn

Abstract

全排列生成算法作为组合论中的重要内容在诸多领域中应用广泛。在本文中，我们针对经典的全排列生成算法：字典序法和 SJT 算法，进行了复杂度分析，证明了两种方法的复杂度均为 $O(n!)$；我们在经典的全排列生成算法基础上，考虑到全排列的对称性的结构特征，实现了全排列生成的加速算法；进一步我们还采用了用位置全排列来替代元素全排列的策略，通过牺牲一部分内存进一步削减了时间开销，通过实验验证该算法可以极大的减少程序运行时间，能够应用于对时间开销要求较高的全排列生成场景中。代码实现地址：https://github.com/JiayiGuo821/FullPermutation。

1 简介

关于排列的生成问题是组合论中的重要内容之一。迄今为止，关于生成全排列的算法已有不少研究成果，Wells (1961) 提出了一种新的基于全排列生成过程中有趣的特点生成全排列的方法，即每个置换都是通过其两个标记的一次互换从其前身衍生而来的。在一半以上的情况下，两个标记是相邻的；Johnson (1963) 提出了一种基于简单规则的全排列方法，通过在相邻位置对两个标记进行一次互换来从其前身推导每个排列；Ginsburg (2007) 根据其约简集合确定一个排列；Lipski (2007) 提出邻位对换法进行全排列；Akl et al. (1994) 提出了一种收缩算法，即用于按字典顺序生成 n 个元素的所有排列，在具有恒定大小内存处理器的线性阵列上执行，每个处理器负责产生给定排列的一个元素。该算法在最弱的并行计算模型上运行可以保证成本最佳并且可以被扩展为自适应地运行；Monks (2009) 提出了一种基于车轮循环的方式进行全排列；Steinhardt (2010) 根据全排列的循环结构和下降集设计全排列生成算法。通过将经典的 Gessel 和 Reutenauer 双射泛化，以处理具有某些上升和下降块的全排列；针对算法运行时间长，难于并行化的缺点，Thanh (2012) 基于反演向量以及子序列构造了一种更简单且易于并行化的算法；在现有的多项式全排列算法的基础上，Akbary et al. (2011) 针对一大类多项式全排列提出了统一的处理方法，产生了构造的几个新的和旧的多项式全排列类；Dershowitz (1975) 提出了一种简单的无环算法生成一组元素的所有排列，并证明了其有效性，其简化了 Ehrlich 的 Johnson and Trotter 算法的无环版本，通过交换先前排

列的两个相邻元素来生成每个排列，通过简单的数据结构避免了在每个连续排列的生成过程中进行循环的需要。

对于多重集的排列问题，一些高效可行的算法被提出。Takaoka (1999) 设计了一种使用数组的数据结构在从排列到排列生成多重集排列的算法；Vajnovszki (2003) 提出了一种用于组合结构的构造器，用于从相似结果中为更简单的对象获得一种新的无环生成算法，用于多重集的全排列；Korsh & Lipschutz (1997) 提出了一种全排列之间需要固定时间的多重集排列算法；受到并行计算的启发，全排列的并行算法引起了很多学者的重视：Krattenthaler (2001) 提出了 132 个避免全排列和 Dyck 路径之间的双射。基于该双射，对 132 个避免全排列进行精确的渐近估计；Akl et al. (1994) 提出了一种在简单的计算模型上执行生成所有全排列的最优并行算法，该模型是具有 n 个相同处理器的线性阵列。由于处理器的简单性和规则性，该模型非常适合 VLSI 实现并且可以以最小的更改顺序生成所有排列；Akl & Stojmenovic (1992) 提出了一种简单的并行全排列生成算法，其在 n 个处理器的线性阵列上执行，每个处理器具有恒定大小的内存，每个处理器负责产生给定排列的一个元素，该算法是成本最优的。Akl (1987) 针对排列组合的问题，描述了三种新的自适应且成本最优的并行算法。

全排列生成算法在计算机科学的许多领域中应用广泛，例如调度问题，系统控制，数据挖掘，图像处理等方面。许多基于混沌的加密方法利用了混沌信号逐渐复杂的行为，Abir & Abdelhakim (2010) 通过比较分段线性混沌映射（PWLCM）和受新技术干扰的 PWLCM 的性能，将两个混沌映射图用于控制具有良好加密特性的三种全排列方法，将其应用于图像处理，由扰动图控制的拟议混沌置换方法表现出更高的性能；Wang et al. (2012) 通过引入基于排列的线性表示法来以易于进化的形式来表达故事，其依赖于捕获基因组中故事中的事件，并且通过实验客观指标评估故事；为了能够实现整数分拆有序和无序的全部排列和组合,杜瑞卿 & 刘广亮 (2013) 详细论证了算法数学原理以及实现的算法描述，为这全排列实际应用提供了理论和方法；李模刚 et al. (2010) 剖析了全排列生成的相关算法，结合克莱姆法则，提出工程应用中线性方程组求解的求解的一个具体思路，为相关工程应用提供一种数值计算方法；徐志才 & 徐志勇 (1984) 提出用图论来解决排列问题，阐明了 Hamilton 回路与 n 个元素的所有排列的关系，给出了全排列生成的 Hamilton 回路法。

在本文中，我们首先列举了两种经典的全排列生成算法：字典序法和 SJT 算法，对两者进行了复杂度分析，证明了两种方法的复杂度均为 $O(n!)$；然后，在经典的全排列生成算法基础上，我们针对全排列总体对称性的结构特征，提出了一种基于对称性的全排列生成加速算法，实现了时间开销的减半。更进一步，当我们选择开辟一定内存来保存排列规则时，可以更多地节省程序的时间开销；最后，我们通过实验验证验证了加速算法的有效性，该算法能够应用于对时间开销要求较高的全排列生成场景中。

2 字典序全排列生成算法复杂度分析

Algorithm 1 字典序全排列生成算法

Input: 正整数 n

Output: 全排列

1 初始化全排列 $p_1p_2\cdots p_n = 12\cdots n$;

2 **do**

3 从新排列的右端开始, 线性搜索第一个比右边数字小的数字的序号 $k-1$, 即 $k-1 = max\{i|p_i < p_{i+1}\}$;

4 在 p_{k-1} 的右边的数字中，搜索出所有比 p_{k-1} 大的数字中最小的数字 p_j，即 $j = min\{i|p_i > p_{k-1}\}$;

5 交换 p_j, p_{k-1};

6 再将排列右端的递减部分 $p_kp_{k+1}\cdots p_n$ 倒转，得到新排列 $p_1p_2\cdots p_{k-2}p_jp_n\cdots p_{k+1}p_k$;

7 **while** *k-1 存在*;

由字典序全排列生成算法，可以 k, p_k 为指标划分生成的全排列集合并计算复杂度，具体地，全排列集合的划分为

$$\sum_{k=1}^{n-1} A_{n-k-1}^{n-k-1} \sum_{p_k=k+1}^{n} C_{p_k-1}^{1} C_{p_k-2}^{k-1} = n! - 1 \tag{1}$$

其中 $C_{p_k-1}^{1}$ 表示 p_{k-1} 的选取，$C_{p_k-2}^{k-1}$ 表示 $p_{k+1}p_{k+2}\cdots p_n$ 的选取（其有序），A_{n-k-1}^{n-k-1} 表示 $p_1p_2\cdots p_{k-2}$ 的选取（其无序）。则上述求和为所有 k-1 存在的排列的数目和，即不包括令算法终止的排列，即为 $n! - 1$。

下面考虑算法的复杂度，对固定的 k，算法中步骤 3 的操作数为 k；对于步骤 4，若线性搜索，由 p_{k-1} 的选取是均匀的，则平均操作数为 $k/2$，注意 p_{k-1} 的右边的数字有序，若采用二分查找，则平均操作数为 log_2k，为了方便计算我们用线性搜索进行估

计；对于步骤 5，操作数为 3；对于步骤 6，操作数为 $k/2$。则总操作数 $S(n)$ 为

$$
\begin{aligned}
S(n) &= n + \sum_{k=1}^{n-1}(k + log_2 k + 3 + \frac{k}{2}) A_{n-k-1}^{n-k-1} \sum_{p_k=k+1}^{n} C_{p_k-1}^{1} C_{p_k-2}^{k-1} \\
&= n + \sum_{k=1}^{n-1}(\frac{3k}{2} + log_2 k) A_{n-k-1}^{n-k-1} \sum_{i=k+1}^{n} C_{i-1}^{1} C_{i-2}^{k-1} + 3(n! - 1) \\
&\leq n + \sum_{k=1}^{n-1}(\frac{3k}{2} + \frac{k}{2}) A_{n-k-1}^{n-k-1} \sum_{i=k+1}^{n} C_{n-1}^{1} C_{n-2}^{k-1} + 3n! \\
&= n!\frac{1}{(n-1)!} + n! \sum_{k=1}^{n-1} \frac{2k}{n} \frac{n-k}{(k-1)!} + 3n! \\
&= n!\frac{1}{(n-1)!} + n! \sum_{k=0}^{n-2} 2\frac{k+1}{n} \frac{n-k-1}{k!} + 3n! \\
&\leq n! + n! \sum_{k=0}^{n-2} 2\frac{k+1}{k!} + 3n! \\
&= n! + n! \sum_{k=0}^{n-2} 2\frac{1}{k!} + n! \sum_{k=0}^{n-3} 2\frac{1}{k!} + 3n! \\
&\leq n!4(e+1) \\
&\approx 14.873n!
\end{aligned}
\tag{2}
$$

由以上结果，对于任意 n 我们给出了总操作数的上界 $14.873n!$，与一般估计 $O(n \times n!)$ 相比，我们的分析表明，对于任意大的 n，平均每个排列生成的操作数由常数控制，进一步地，可以利用计算软件计算 $S(n)$ 的值，matlab 可计算的最大值为 $n = 170$，平均操作数约为 6.196。

3 SJT 全排列生成算法复杂度分析

Algorithm 2 SJT 全排列生成算法的一种实现

Input: n

Output: 全排列

初始化全排列 $p_1 p_2 \cdots p_n = 12 \cdots n$，且每个数字的箭头指向均为向左;
初始化三个长度为 n 的列表，分布存储每个数字的位置，箭头指向，及是否可移动;
do
    从可移动状态列表中线性搜索最大的可移动数 m;
    从箭头指向列表及位置列表得到 m 的箭头指向和在排列中的位置;
    交换 m 与其箭头指向的相邻数字并更新二者的位置;
    将全部比 m 大的数字的箭头转向及置为可移动状态;
while m 存在;

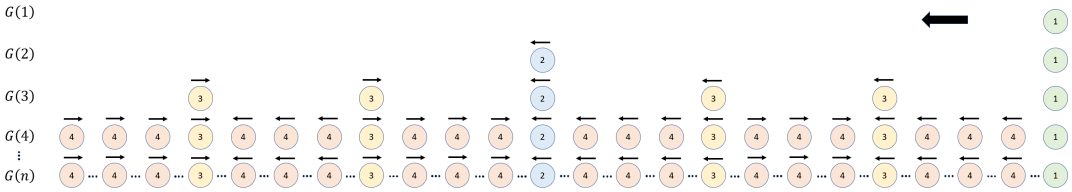

图 1: SJT 算法步骤

如图所示，SJT 算法步骤图中每行表示 n 个元素的全排列 $G(n)$ 的 SJT 算法生成过程，顺序为从右至左，其中 1 表示初始排列，数字 $i, i \geq 1$ 及箭头方向表示该序列由 i 依箭头方向移动得到。

由上图 SJT 算法步骤可视化，SJT 算法实质上是对简单递归算法的改进循环版本，通过箭头指向移动的特性保证了生成排列的连续性。从步骤图中可以直观的看出，当数字 m 移动前，任意比 m 大的数字 i 均不可移动，达到了可移动的边界，即去掉排列中比 i 大的数字并保持原序组成新的排列，则 i 位于排列的两端，综合来看，所有比 m 大的数字均位于排列的两端，且越靠近两端的数字越大。此外，数字 m 移动后，任意比 m 大的数字箭头转向后全部变为可移动。

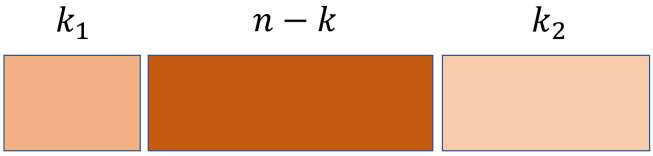

图 2: 任一排列

由上述分析，在数字 m 移动前，令 $n-k=m, k_1+k_2=k$，排列的状态如图所示，即将排列分为三部分，两端为比 m 大的数字，中间为 $1, 2, \cdots, m$ 组成的子序列，且 m 不在子序列的两端，否则其不可移动。可以 $k, k2$ 为指标划分生成的全排列集合并计算复杂度，具体地，全排列集合的划分为

$$\sum_{k=0}^{n-3}\sum_{k_2=0}^{k} C_k^{k_2} A_{n-k-1}^{n-k-1} C_{n-k-2}^{1} = n! - 2^{n-1} \tag{3}$$

其中 $C_k^{k_2}$ 表示排列右端 k_2 长度子列的选取 (其有序)，$A_{n-k-1}^{n-k-1} C_{n-k-2}^{1}$ 表示排列中间 $n-k$ 长度子列的选取 (其无序，且 m 不在子序列的两端)，剩余数字位于排列左端 k_1 长度的子列 (其有序)。则上述求和为所有 m 不在中间 $n-k$ 长度子序列的两端的排列的数目和，即 $m \geq 3$。当数字 2 移动时，对应的排列数为 $\sum_{k_2=0}^{n-2} C_k^{k_2} A_{n-k}^{n-k} = 2^{n-1}$。符合上述划分结果。

下面考虑算法的复杂度，对固定的 k，算法中步骤 4 的操作数为 $k+1$；对于步骤 5，操作数为 2；对于步骤 6，操作数为 5，其中交换的操作数为 3；对于步骤 7，操作

数为 $2k$。对于 $n \geq 3$ 则总操作数 $S(n)$ 为

$$\begin{aligned}
S(n) &= 2^{n-1}(3(n-2)+11) + \sum_{k=0}^{n-3}(3k+11)\sum_{k_2=0}^{k} C_k^{k_2} A_{n-k-1}^{n-k-1} C_{n-k-2}^{1} \\
&= 2^{n-1}(3(n-2)+11) + 11(n!-2^{n-1}) + \sum_{k=0}^{n-3} 3k2^k(n-k-1)!(n-k-2) \\
&= 2^{n-1}(3n-6)) + 11n! + n!\sum_{k=0}^{n-3}\frac{2^k}{k!}\frac{(n-k)!k!}{n!}\frac{n-k-2}{n-k}3k \\
&\leq 3n!\frac{2^{n-1}}{(n-1)!} + 11n! + n!\sum_{k=0}^{n-3}\frac{2^k}{k!}\frac{3k}{C_n^k} \\
(\text{易得}\frac{3k}{C_n^k} \leq 1) &\leq 3n!\frac{2^{3-1}}{(3-1)!} + 11n! + n!\sum_{k=0}^{n-3}\frac{2^k}{k!} \\
&\leq n!(17+e^2) \\
&\approx 24.390n!
\end{aligned} \tag{4}$$

由以上结果，对于任意 n 我们给出了总操作数的上界 $24.390n!$(显然对 $n = 1, 2$ 是也满足)，利用 matlab 计算 $n = 170$ 时平均操作数约为 11.036。

我们实现的 SJT 算法特点在于维护每个数字的位置列表及是否可移动的状态，事实上若不维护可移动列表，步骤 4 可通过交替搜索排列两端的数字找到最大的可移动数，其操作数为 $O(k)$，平均操作数仍为常数，但略增长。但是若不维护位置列表，则步骤 4 确定最大可移动数在序列中的位置的平均操作数为 $(n-k-1)/2$，即为线性搜索开销，这将导致最终平均操作数约为 $n/2$。

4 无重排列的对称性与基于对称性的全排列生成加速算法

对于全排列生成问题，一个重要的缺陷就是时间开销大。计算一个长度为 n 的无重全排列通常需要 $O(n!)$ 甚至是 $O(n\times n!)$ 的时间复杂度。如何有效地降低生成全排列的时间开销，是本节重点讨论的问题。在本节中，我们首先探究了全排列生成中的对称性，并据此提出了基于对称性的全排列生成加速算法。理论分析和代码实验测试均表明，在生成同样长度的全排列的要求下，基于对称性的全排列生成加速算法可以极大地降低时间开销。

4.1 全排列的对称性

对于任意长度为 $n(n \geq 2)$ 的无重排列，可知全排列的总数为 $n!$。由 $n \geq 2$ 可知，$n!$ 一定为偶数 $(2k)$。结合无重排列的特性可知，对于全排列中的任意给定排列，都能且仅能找到一个与之互为倒序的排列。图3给出了全排列的对称性说明。

根据全排列的对称性，如果能找到一种算法，可以输出前 $\frac{n!}{2}$ 个互不为逆序的排列，则可以对其中每一个生成的排列做一次逆序得到另一半的排列。由于将数组顺序逆序

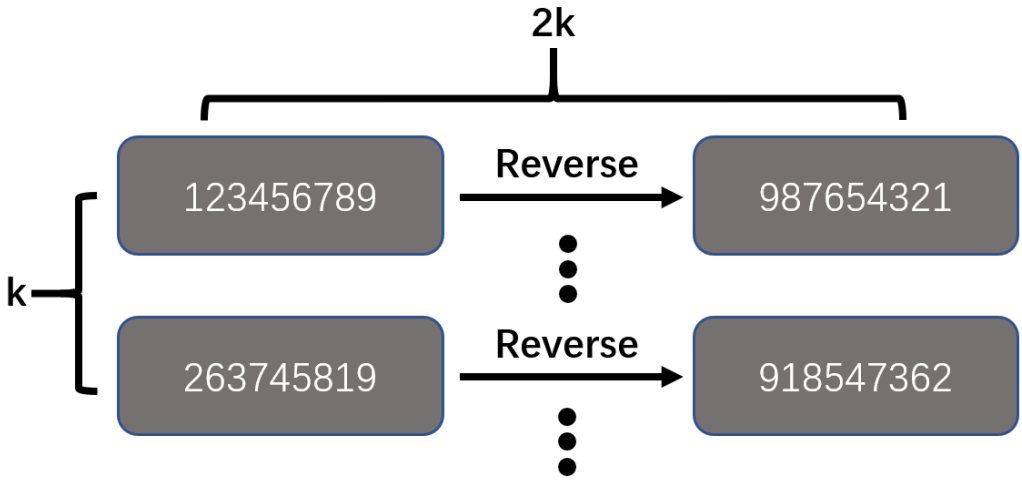

图 3: 全排列的对称性说明

输出的时间远小于单个排列生成的时间（譬如由序号到中介数再到排列的生成模式），故基于对称性的方法可以近似节省一半的时间开销。

4.2 基于对称性的全排列生成加速算法

对于一个长度为 n 的全排列，如何得到前 $\frac{n!}{2}$ 个互不为逆序的排列？本小节给出一种简明有效的方法。长度为 n 的排列可以表示为 $a_1a_2a_3...a_n$，其中 a_1 为首位，a_n 为末位。简明的讲只要控制好 a_1 和 a_n 的取值就可以实现前述的目标了。

在实际操作中，首先从 1 到 $n-1$ 遍历 a_1，对于每一个确定的 a_1，从 a_1+1 到 n 遍历 a_n。对于剩下的 $n-2$ 位用任意的方法进行全排列生成即可。这是因为若当前遍历的首位为 i，则末尾至少为 $i+1$，因此该序列的逆序一定未在之前的遍历中出现过。故一共需遍历 $\frac{n(n-1)}{2}$ 组不同的 a_1 和 a_n，每一组 a_1 和 a_n 对应 $(n-2)!$ 个序列，所以共生成了 $\frac{n!}{2}$ 个互不为逆序的排列。对于长度为 n 的全排列，有且仅有 $\frac{n!}{2}$ 个互不为逆序的全排列，所以对这 $\frac{n!}{2}$ 个排列逆序输出一次即可得到全排列。算法3对给出了上述流程的伪代码。

Algorithm 3 基于对称性的全排列生成算法

设排列长度为 n，排列表示形式为 $a_1, a_2, ...a_n$，任意全排列生成算法 $FP()$。
for $a_1 = 1$ **to** $n-1$ **do**
  for $a_n = a_1 + 1$ **to** n **do**
    设剩余的 $n-2$ 个数字为 $b_1, b_2, ...b_{n-2}$
    for $i = 1$ **to** $(n-2)!$ **do**
      $a_{2_i}, a_{3_i}, ...a_{(n-1)_i} = FP(b_1, b_2, ...b_{n-1})_i$
      顺序序列为 $a_1, a_{2_i}, a_{3_i}, ...a_{(n-1)_i}, a_n$
      逆序序列为 $a_n, a_{(n-1)_i}, a_{(n-2)_i}, ...a_{2_i}, a_1$
    end for
  end for
end for

4.3 牺牲内存的改进加速算法

对于上述算法还可以进一步加速。观察到算法3中时间开销主要集中在全排列生成算法上。而事实上对于任意 $n-2$ 个大小不同的数字，其全排列的结果规则都是一致的。举例说明，对于 $1,2,...n-2$ 的一个排列，1 所出现的位置事实上也代表的是 $n-2$ 个大小不同的数字中最小的一个所在的位置，同理任意数字 i 出现的位置也代表着从小到大第 i 个数字出现的位置。因此上述算法中仅需生成一次 $1,2,...n-2$ 的全排列作为规则存储在内存中，每次生成长度为 $n-2$ 的全排列是对该规则调用。一种简单的规则调用模式是来，从小到大将待排数字存在长度为 $n-2$ 数组中即可，全排列规则对应的正是数组的索引。算法4给出了上述流程的伪代码。

Algorithm 4 基于对称性的全排列生成算法

设排列长度为 n，排列表示形式为 $a_1,a_2,...a_n$，任意全排列生成算法 $FP()$。
$rule = FP(1,2,...n-2)$
for $a_1 = 1$ **to** $n-1$ **do**
  for $a_n = a_1 + 1$ **to** n **do**
    设剩余的 $n-2$ 个数字从小到大依次为 $b_1,b_2,...b_{n-2}$
    for $i = 1$ **to** $(n-2)!$ **do**
      $a_{2_i},a_{3_i},...a_{(n-1)_i} = (b_1,b_2,...b_{n-1})_{rule_i}$
      顺序序列为 $a_1,a_{2_i},a_{3_i},...a_{(n-1)_i},a_n$
      逆序序列为 $a_n,a_{(n-1)_i},a_{(n-2)_i},...a_{2_i},a_1$
    end for
  end for
end for

上述改进可以加速基于对称性的全排列生成算法，但同时由于需要存储 $(n-2)!$ 条排列规则，内存开销加剧，实际上这是一种牺牲空间开销来节省时间开销的改进方式。

5 实验

在本节中，我们对基于对称性的全排列生成加速算法进行了时间开销的 C++ 代码测试，并与原始算法进行了对比。

基于对称性的全排列生成实际上是对原有算法的加速，故这里对比了四种算法（字典序法、递增进位制数法、递减进位制数法和邻位对换法）在基于对称性的全排列加速前后的时间开销对比，以及牺牲内存开销进一步节省时间开销的情况。原生的全排列实现方式均为遍历序号数生成中介数最后得到排列。

从表1和表2中结果可以看到，基于对称性加速后，时间开销大约可以减少一半，而在牺牲了内存来存储长度为 $n-2$ 的全排列规则后，时间开销更是被极大的削减。这与我们的理论分析是一致的。

表 1: 基于对称性的八位全排列生成与原全排列生成算法的时间开销对比

	字典序法	递增进位制数法	递减进位制数法	邻位对换法
原始算法	2781ms	2375ms	2375ms	3344ms
基于对称性加速	1266ms	1250ms	1281ms	1594ms
牺牲内存进一步加速	406ms	406ms	406ms	422ms

表 2: 基于对称性的九位全排列生成与原全排列生成算法的时间开销对比

	字典序法	递增进位制数法	递减进位制数法	邻位对换法
原始算法	30203ms	24594ms	24781ms	33969ms
基于对称性加速	13938ms	12875ms	12891ms	16750ms
牺牲内存进一步加速	4063ms	4078ms	4203ms	4125ms

由此不禁让人想到 $n-2$ 位的排列可以继续化简为对称情况下的 $n-4$ 位的排列，如此递归下去，最终时间开销是不是趋于 0 了呢? 答案是否定的，对于新算法的在时间上的节约，本文始终称之为时间开销上的减少，而并未称之为时间复杂度上的减少。这是因为算法的操作数并没有发生变化，始终需要输出 $n!$ 个排列。对于计算机而言，由于物理地址固定，正序和逆序输出同一个数组是高效的，而从序号数得到排列的过程是冗长的。故在本算法下逆序输出的时间可以忽略不记。但依上述递归排列长度从 $n-2$ 减小到 $n-2k$。实际上正序逆序这种翻转的操作数也增大了 k 倍。k 越大，正序逆序翻转所带来的时间开销越不可忽略，同时需要遍历的循环数也越来越多，故时间开销是不会随着递归一直减小的。

6 结论

本文分析了字典序法和 SJT 算法的时间复杂度，证明了两种方法的复杂度均为 $O(n!)$，提出了一种高效的基于对称性的全排列生成加速算法，通过数值分析与实验测试验证了算法的有效性和高效性。算法的创新之处在于：考虑了全排列的结构特征，充分利用对称性这一特点，可以极大的节省全排列的生成时间；基于位置排列来替代元素全排列的思路，我们进一步改善了算法性能，极大地节省了程序运行的时间。该加速算法可以应用于需要在较短时间内进行全排列的场景中。

References

Awad Abir and Saadane Abdelhakim. Efficient chaotic permutations for image encryption algorithms. *Lecture Notes in Engineering Computer ence*, 2183(1), 2010.

Amir Akbary, Dragos Ghioca, and Qiang Wang. On constructing permutations of finite fields. *Finite Fields Thr Applications*, 17(1):51–67, 2011.

Selim G Akl. Adaptive and optimal parallel algorithms for enumerating permutations and combinations. *The Computer Journal*, 1987.

Selim G. Akl and Ivan Stojmenovic. A simple optimal systolic algorithm for generating permutations. *Parallel Processing Letters*, 2(2n03):–, 1992.

Selim G. Akl, Henk Meijer, and Ivan Stojmenovic. An optimal systolic algorithm for generating permutations in lexicographic order. *Journal of Parallel and Distributed Computing*, 20(1):84–91, 1994.

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

John Ginsburg. Determining a permutation from its set of reductions. *Ars Combinatoria*, 82:55–68, 2007.

Selmer M Johnson. Generation of permutations by adjacent transposition. *Mathematics of computation*, 17(83):282–285, 1963.

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

C. Krattenthaler. Permutations with restricted patterns and dyck paths. *Advances in Applied Mathematics*, 27(2-3):510–530, 2001.

Witold Lipski. Kombinatoryka dla programistów. *Wydawnictwa Naukowo-Techniczne (WNT), Warsaw*, 2007.

Maria Monks. Reconstructing permutations from cycle minors. *Electronic Journal of Combinatorics*, 16(1), 2009.

Jacob Steinhardt. Permutations with ascending and descending blocks. *Electronic Journal of Combinatorics*, 17(1):1110–1119, 2010.

Tadao Takaoka. An o(1) time algorithm for generating multiset permutations. In *Algorithms and Computation, 10th International Symposium, ISAAC '99, Chennai, India, December 16-18, 1999, Proceedings*, 1999.

Hoang Chi Thanh. Parallel generation of permutations by inversion vectors. In *2012 IEEE RIVF International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future*, pp. 1–4. IEEE, 2012.

Vincent Vajnovszki. A loopless algorithm for generating the permutations of a multiset. *Theoretical Computer ence*, 307(2):415–431, 2003.

Kun Wang, Vinh Bui, Eleni Petraki, and Hussein A. Abbass. From subjective to objective metrics for evolutionary story narration using event permutations. 2012.

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

徐志才 and 徐志勇. 产生 n 个元素全排列的 hamilton 回路法. 北京邮电学院学报, (03): 111–115, 1984.

李模刚, LI, Mo-gangDepartment, of, Computer, , , Control, Engneering, Lanzhou, and Petrochemical and. 全排列生成算法在克莱姆法则中的应用. 现代计算机 *(专业版)*, 09(No.236):15–17, 2010.

杜瑞卿 and 刘广亮. 整数分拆以及等差数列多重约束条件下全排列的生成法. 2013.