全排列生成算法的研究与优化

秦雪迪*、仝美涵†、田冰‡
清华大学计算机科学与技术系软件所

摘要

全排列生成问题指的是对于 n 个相互不同的元素，如何输出这 n 个元素所有不同的排列。本文研究并实现了字典序法、利用中介数产生全排列的字典序法、递增进位制法、递减进位制法以及邻位对换法，并通过实验对这五种算法的效率和复杂度进行了分析。同时，在这五种算法的基础上，结合动态规划和二叉树的思想，提出了三种优化求解中介数的算法，同时利用中介数未进位的特殊性，对生成后续排列的过程进行了优化。实验结果表明，我们的优化能够显著提高全排列算法的性能。

1 引言

组合数学是一门研究离散对象的科学，是计算机科学的基础。随着计算机的应用越来越普及，组合数学的研究将更加深入。其最主要的研究内容就是对离散对象的计数 [4]。学习组合数学的基础就是排列组合。排列是指从多个不同元素中取出几个元素按照一定顺序排成一列的过程，排列的种类数称为排列数，用 P_n^m 表示。组合是指从多个不同元素中取出元素组成一个集合的过程，组合的种类数称为组合数，用 C_n^m 表示。全排列的生成算法是将给定的序列中所有可能的全排列无重复无遗漏地枚举出来。此处全排列的定义是：从 n 个元素中取出 m 个元素进行排列，当 $n = m$ 时这个排列被称为全排列。字典序、邻位对换法、递增进位制法、递减进位制法都是常见的全排列生成算法。全排列生成算法有广泛的应用，例如可以应用在不同的组合优化问题（例如 TSP 问题）、n 皇后问题、克莱姆法则 [5]、图论问题（例如图同构问题）等的算法设计中 [2]。针对全排列的生成，Robert Sedgewick 在 1997 年写了一份全排列生成算法的调查报告 [3], 对当时几乎全部的全排列生成算法做了比较，并得出一个结论：Heap 提出的全排列生成算法在大多数情况下是最快的。C++ 的 STL 中定义的 next_permutation 和 prev_permutation 函数也是十分灵活高效的算法，被广泛应用于为指定序列生成后一个和前一个排列。除此之外，还有一类利用中介数和排列一一对应的关系，从而利用中介数生成全排列的算法，例如，字典序全排列，递增进位制法，递减进位制法以及邻位对换法等。

本文即是针对这几种算法，进行研究和改进。本文的工作如下：

- 实现了五个全排列生成算法同时对算法效率和复杂度进行了分析;
- 利用中介数未进位的特殊性，对生成后续排列的过程进行了优化，并通过实验比较优化前后的算法效率;

*秦雪迪 2017310722 qxd17@mails.tsinghua.edu.cn
†仝美涵 2017310729 tongmh17@mails.tsinghua.edu.cn
‡田冰 2017310757 tb17@mails.tsinghua.edu.cn

- 利用基于动态规划和不同特点的二叉树的思想，提出了三种优化求解中介数的算法，并通过实验比较优化前后的算法效率；

2 相关工作

2.1 字典序法

设 P 是 $1 \sim n$ 的一个全排列，$P = p_1 p_2 \cdots p_n$。字典序法就是按照字典序依次求出下一个排列的算法。这要求相邻的两个字典有尽可能长的共同前缀，将变化限制在尽可能短的后缀上。具体方法如下：

1. 从排列的右端开始，找出第一个比右边数字小的数字的位置 j
2. 找出在 p_j 右边的数字中，比 p_j 大的数中最小的数字 p_k
3. 交换 p_j 和 p_k，将序列 $p_{k+1} \sim p_n$ 倒转即可以得到下一个序列

除此之外，字典序法也可以通过中介数进行计算。中介数是计算排列的中介环节，它的每一位由原排列每个数字右侧比其小的数字个数构成。其运算过程为：原排列 $\rightarrow$ 中介数 $\rightarrow$ 新中介数 $\rightarrow$ 新排列。

设得到的新中介数位 $b = b_n \cdots b_2$，要求的排列为 $r = r_n \cdots r_1$。

算法流程： 中介数初始值每一位都是 0，根据中介数，从高位到低位依次遍历中介数从而确定排列的每一位，若当前中介数中第 i 位的值为 b_i，就在剩余的还没有被排过的数中找出第 $b_i + 1$ 小的数，那么 r_i 就等于这个数。依次遍历完中介数的每一位后，还没被排过的数即为 r_1 位对应的数。由此，一个新的排列生成。中介数加 1，根据上述算法产生下一个排列。直到中介数最高位等于产生进位，算法结束，所有的字典序全排列都已生成。

2.2 递增进位制法

人们通常所用的进制大多是固定进位制，如 2 进制，10 进制等。m 位 n 进制可以表示的数字个数为 m^n 个。而递增/递减进位制数，顾名思义，是指数字的进制随着数字位置的不同递增/递减的进制。m 位递增/递减进位制数可以表示 $m!$ 数字。例如递增进位制 4121，它的进制从右向左依次为 2，3，4，5，换算成十进制相当于数字 107。

递增进位制法生成全排列可以通过中介数进行计算。设中介数位 $b = b_n \cdots b_2$，b_i 表示在排列中 i 这个数字右边比 i 小的数有 b_i 个，假设要求的排列为 $r = r_n \cdots r_1$。

算法流程： 中介数初始值每一位都是 0，根据中介数，从高位到低位依次遍历中介数从而确定排列的每一位，若当前中介数中第 i 位的值为 b_i，则从右向左扫描所有没有被排过数字的位置，直到第 $b_i + 1$ 个位置 j，那么 i 就在这个位置上，即 $r_j = i$。依次遍历完中介数的每一位后，还没被排过的位置即为 1 所在的位置。由此，一个新的排列生成。中介数加 1，根据上述算法产生下一个排列。由此，一个新的排列生成。中介数按照递增进位制加法加 1，根据上述算法产生下一个排列。直到中介数最高位等于产生进位，算法结束，所有的递增进位制法全排列都已生成。

2.3 递减进位制法

在通过递增进位制的加法生成下一个中介数时，由于递增进位制越低位进制越小，所以在加法的过程中极易产生进位，算法效率受到了限制，因此，提出了一种新的算法：递减进位制法。和递增进位制法不同，递减进位制越高位进制越小。在通过中介数生成全排列时，算法流程与递增进位制法类似。

算法流程： 设中介数位 $b = b_2 \cdots b_n$，b_i 表示在排列中 i 这个数字右边比 i 小的数有 b_i 个，假设要求的排列为 $r = r_n \cdots r_1$。中介数初始值每一位都是 0，根据

中介数，从低位到高位依次遍历中介数从而确定排列的每一位，若当前中介数中第 i 位的值为 b_i，则从右向左扫描所有没有被排过数字的位置，直到第 b_i+1 个位置 j，那么 i 就在这个位置上，即 $r_j=i$。依次遍历完中介数的每一位后，还没被排过的位置即为 1 所在的位置。由此，一个新的排列生成。中介数按照递增进位制加法加 1，根据上述算法产生下一个排列。直到中介数最高位等于产生进位，算法结束，所有的递减进位制法全排列都已生成。

2.4 邻位对换法

递增进位制和递减进位制数字的换位是单向的，而邻位对换法的换位是双向的，通过保存数字的 “方向性” 来快速得到下一个排列。

算法流程： 设中介数位 $b=b_2\cdots b_n$，b_i 表示在排列中 i 这个数字右边比 i 小的数有 b_i 个，假设要求的排列为 $r=r_n\cdots r_1$。由中介数得到排列时，假设第 b_k 位的值为 i，方向是向左的，就从右向左扫描所有没有被占过的位置，知道第 b_i+1 位置 j，那么 i 就在这个位置上，即 $r_j=i$。如果方向是向右的，那么转化成向左的进行计算，即表示 i 的左边比 i 小的数有 b_i 个也即相当于 i 的右边比 i 小的数有 $i-1-b_i$ 个。依次遍历完中介数的每一位后，还没被排过的位置即为 1 所在的位置。由此，一个新的排列生成。中介数按照递增进位制加法加 1，根据上述算法产生下一个排列。直到中介数最高位等于产生进位，算法结束，所有的临位对换法全排列都已生成。

3 算法效率和时间复杂度分析

3.1 时间复杂度分析

n 个数字的全排列一共有 $n!$ 种，所以全排列算法至少是 $O(n!)$ 的。字典序算法在生成下一个排列时需要扫描全排列找出第一个下降的数字并交换，因此字典序算法的时间复杂度是 $O(n*n!)$；采用中介数的字典序法、递增进位制数法、递减进位制数法和邻位对换法在求下一个全排列时，需要先生成中介数，而生成中介数的时间复杂度是 $O(n^2)$，因此它们的时间复杂度是 $O(n^2*n!)$。所以实际上，全排列算法对大型的数据是无法处理的。也正因为如此，研究如何降低全排列生成算法的时间复杂度是非常重要的。

3.2 几种全排列生成算法的效率分析

本节我们比较了字典序算法（DIC1）以及基于中介数的字典序算法（DIC2）、递增进位制法（INCRE）、递减进位制法（DIM）以及邻位对换法（EXCHANGE）这五种算法在生成全部排列方面的性能，如图 1 所示。

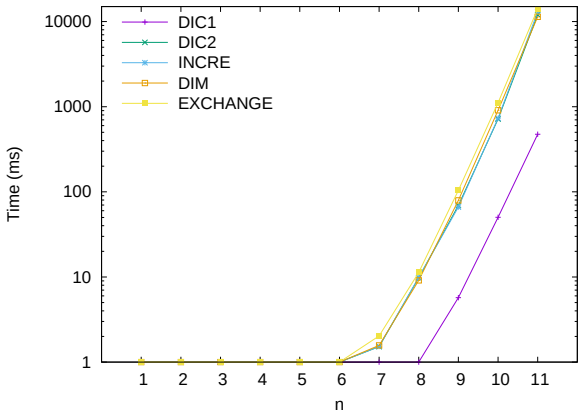


图 1: 全排列生成算法比较

从图中可以看出，不使用中介数的字典序算法是五种算法中最快的算法，而使用中介数的邻位对换法是五种算法中最慢的算法。当排列的位数超过 10 后，使用中介数的四种算法性能急剧下降，这是因为由排列得到中介数和由中介数得到排列的计算次数急剧增加。可见当位数较大时，使用中介数作为媒介会大大降低全排列生成算法的性能。

4 利用中介数未进位的特殊性生成后续排列

4.1 特殊性分析

在四种算法中，如若我们希望求当前排列数 P 的后第 A 个排列数, 在中介数没有发生进位的情况下，我们可以通过移动原排列数的若干位，直接生成新的排列数。

设 P 是 $1 \sim n$ 的一个全排列，$P = p_1p_2 \cdots p_n$，数组 I 存储排列中每个数字的下标，则 I_i 表示数字 i 在排列 P 中的位置。定义 C 为:

$$C = c_1c_2 \cdots c_n$$

$$c_i = |\{x < i \mid x \in \{p_1p_2 \cdots p_{I_i-1}\}| \tag{1}$$

设 A 对应的递减进制下的数为 $a = a_1a_2 \cdots a_{n-1}$，那么若 $\forall i \in [1, n]$，$a_i \leq c_i$ 都成立，则 P 生成新排列的过程中中介数不会发生进位。同理，求排列 P 的前第 A 个排列数时，中介数不进位的条件是 $\forall i \in [1, n]$，$a_i \leq c_i$ 都成立，其中 $c_i = |\{x < i \mid x \in \{p_{I_i+1}p_{I_i+2} \cdots p_n\}|$。

下述算法是在该条件成立下计算新排列数的。

4.2 算法介绍

该算法省去了生成中介数的过程，可以直接根据原排列 P 和改变量 A，求出新的排列数，具体流程如下:

1. 遍历排列数 P, 计算 C, 并将 $2 \sim n$ 所处位置的下标记录在位置数组 I 中
2. 根据选择的算法，将 A 转换位递增进制数或者递减进制数
3. 判断 A 与 C 是否满足公式（1）中所给的条件。若不满足，则终止算法
4. 从后向前遍历 I，假设当前遍历到 I 的第 i 位 I_i
5. 根据 I_i 中记录的位置信息，找到数字 $i+2$ 在原排列中的位置
6. 从数字 $i+2$ 开始，与比其小的数字交换 a_i 次，求出该数字在新排列中的位置
7. 当 i 等于 0 时结束算法

4.3 优化算法性能分析

上述流程中, 前三步的时间复杂度都为 $O(n)$, 后三步时间复杂度近似于 $O(n^2)$, 虽然与传统算法的时间复杂度一致，但是与其相比，该算法省去了生成中介数，再反向生成新排列的过程，在实际的实验中我们可以看到明显的速度提升。

5 基于动态规划生成中介数

基于中介数的四种全排列生成算法的时间复杂度为 $O(n^2 * n!)$，其中求解中介数的时间复杂度为 $O(n^2)$。以字典序为例，当 $n = 100000$ 时，仅求解中介数就需要大约 $24s$，这是十分耗时的。因此本节和下一节提出了两种优化求解中介数的算法：基于动态规划和二叉树生成中介数的算法。本节将详细介绍基于动态规划生成中介数的算法思想。

5.1 原理分析

以字典序为例，通过观察中介数的求解过程，我们得到了两个重要的结论:

1. 当求解排列中数字 i 的中介数时，设 i 左右两侧排列分别为 $P1$，$P2$，若 i 左侧数字个数小于右侧数字个数，即 $|P1| < |P2|$，那么扫描 $P1$ 中比 i 小的数字个数可以更快的得到 i 的中介数;

2. 设 i 与 $i+1$ 之间的排列为 $P3$，若已知 $i+1$ 的中介数，且 i 与 $i+1$ 在 P 中距离很小，那么可以通过 $i+1$ 的中介数快速求解出 i 的中介数；

设 $P = P[1], P[2], ..., P[n]$，数组 A 存储排列中每个数字对应的中介数，即 $A[i]$ 为 $P[i]$ 的中介数；数组 I 存储排列中每个数的下标，即 I_i 表示数字 i 在排列 P 中的位置。综合以上观察，在求解排列中数 i 的中介数时，由于 i 在 P 中的位置为 $I[i]$，故 $P1 = P[1 : I[i]]$，$P2 = P[I[i] + 1 : n]$，$P3 = P[I[i] : I[i + 1]]$，令 $P_{min} = min(P1, P2, P3)$，即 $P1$，$P2$，$P3$ 中最短的一段，通过扫描 P_{min} 中比 i 小的数字个数（设为 t）来计算 i 的中介数，即 $A[I[i]]$，可以有效缩短计算中介数的时间。当 $P1$ 最短时，t 为 i 左侧比 i 小的数字个数，而比 i 小的数字个数共有 $i-1$ 个，故 $A[I[i]] = i-1-t$；当 $P2$ 最短时，$A[I[i]] = t$；当 $P3$ 最短时，若 i 位于 $i+1$ 左侧，即 $P = \cdots i \cdots i+1 \cdots$，由于比 $i+1$ 小的数都会比 i 小，故而 i 的中介数等于 t 加上 $i+1$ 的中介数，若 i 位于 $i+1$ 的右侧，即 $P = \cdots i+1 \cdots i \cdots$，则 i 的中介数等于 $i+1$ 的中介数减去 $1+t$。算法 1 展示了这一过程。

Algorithm 1 DP (P,n)

```
calculate I from P
calculate A[I[n]]
for i ← n − 1 to 1 do
    P1 ← P[1 : I[i]]
    P2 ← P[I[i] + 1 : n]
    P3 ← P[I[i] : I[i + 1]]
    P_min ← min(P1, P2, P3)
    t ← the number of numbers in P_min smaller than i
    if P1 = P_min then
        A[I[i]] ← i − 1 − t
    else if P2 = P_min then
        A[I[i]] ← t
    else
        if I[i] < I[i + 1] then
            A[I[i]] ← A[I[i + 1]] + t
        else
            A[I[i]] ← A[I[i + 1]] − t − 1
        end if
    end if
end for
return A
```

5.2 优化算法性能分析

优化算法每次扫描的排列序列为 $P1$，$P2$，$P3$ 中最短的一段，因此从理论上来说优化算法可以有效减少求中介数的时间。而且当排列较为均匀，即 $\sum_{i=1}^{n-1} |I[i] - I[i+1]|$ 较小时（比如 $P = 1, 2, 3, ..., n$），由于 $P3$ 很小，优化算法可以在接近 $O(n)$ 的时间内求出排列的中介数。

6 利用二叉树生成中介数

6.1 原理分析

当我们从排列数生成中介数时，实际是在求每位数字在排列数中右边比它小的数的个数，那么我们可以从右向左构建二叉树来优化这一过程。设 P 是 $1 \sim n$ 的一个全排列，$P = p_1 p_2 \cdots p_n$，设 A 是 P 对应的中介数，$A = a_1 a_2 \cdots a_{n-1}$。树中每个结点 T 由以下三个元素构成：

d	记录排列数
l	记录左孩子的数目
m	记录排列数对应的中介数

这里我们以递减进位制为例来说明我们的算法，递增进位制，字典序进位制类似。

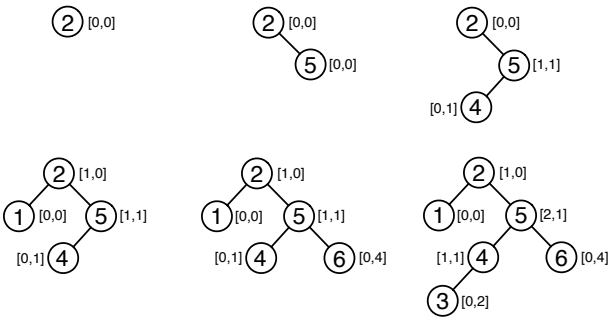


图 2: 利用二叉树生成中介数

我们从 $p_n \sim p_1$ 自后向前依次计算中介数。假设当前计算到 p_i 的中介数 a_{i-1}, 那么我们先利用 p_i 构建一个树的结点 T_i，从树根 T_0 开始，将 T_i 依次与树中的结点比较。假设我们比较到第 k 个结点 T_k, 如果 T_k 中存储的排列数比 T_i 存的小, 则走左岔路，并更新 T_k 左孩子的个数；如果结点 T_k 中存储的排列数比 T_i 存的大，则走右岔路，并更新 T_i 的中介数。一旦 T_i 到达叶结点，则终止算法。其中左孩子和中介数的更新规则为 $l_k = l_k + 1$，$m_i = m_i + l_k + 1$。

图 2 展示了排列 “361452” 生成中介数的过程。其中每个结点存储两个信息，第一个为该节点的左孩子数，第二个为该节点对应的中介数。可以看到每插入一个结点，根据该节点第二个中介数信息，就能够立刻得到该结点对应的中介数。

6.2 利用平衡二叉树进一步优化算法

可以看到，上述算法构建的树并不平衡，这会导致插入效率降低，为了更好的时间性能，我们引用了 AVL 平衡二叉树 [1]。我们将树的节点重新定义如下：

d	记录排列数
l	记录左孩子的数目
r	记录右孩子的数目
a	记录左树的深度
b	记录右树的深度
m	记录排列数对应的中介数

图 3 显示了排列 “361452” 在平衡二叉树上的计算流程。每插入完一个节点，自底向上地采用单旋或者双旋的方法，调整一次树，保证每个节点左右树深之差不超过 1，使得算法的效率得到进一步的提高。

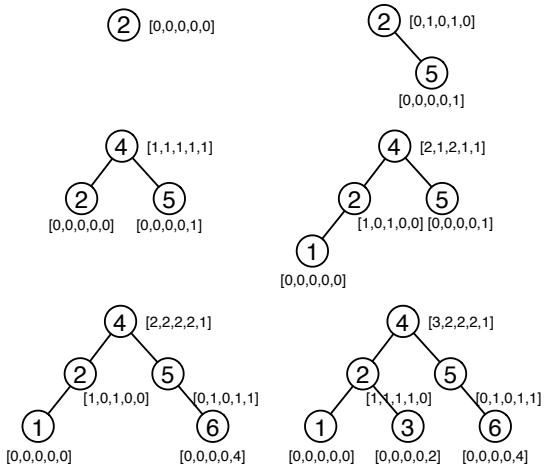


图 3: 利用平衡二叉树生成中介数

6.3 优化算法性能分析

可以看到，我们需要从右向左依次寻找中介数，该过程为 $O(n)$。而寻找中介数的过程实际为 p_i 插入二叉树的过程，该过程的时间复杂度为 $O(log(n))$。因此，最终算法的时间复杂度为 $O(n * log(n))$。可以看到相比于传统算法 $O(n^2)$ 的复杂度，该算法的复杂度更低，后续实验也证实了该算法的时间优越性。

7 实验

为了对优化算法进行性能分析，我们设计了一系列实验来进行测试。本文的所有实验都是在一个 16GB 内存，2.5GHZ 的 Intel Core i7 处理器的 Macbook Pro 上进行的。

7.1 利用中介数未进位的特殊性生成后续排列

我们依次令 n 取不同的值，随机生成排列 P 和一个满足算法条件的正整数 a，分别使用递减进位制算法和优化的算法求解 P 的后第 a 个排列，记录计算时间，得到图 4 。由图可知，优化算法省去了大量不必要的计算，极大缩短了计算时间。

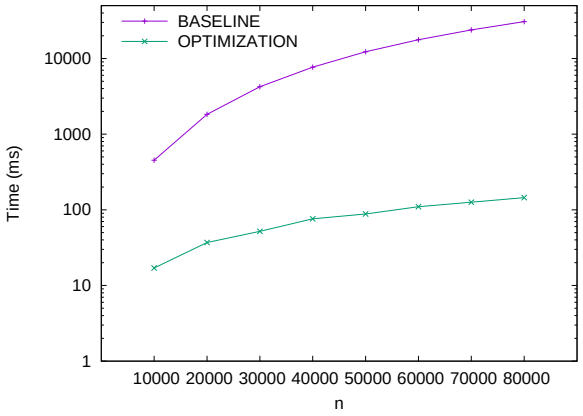
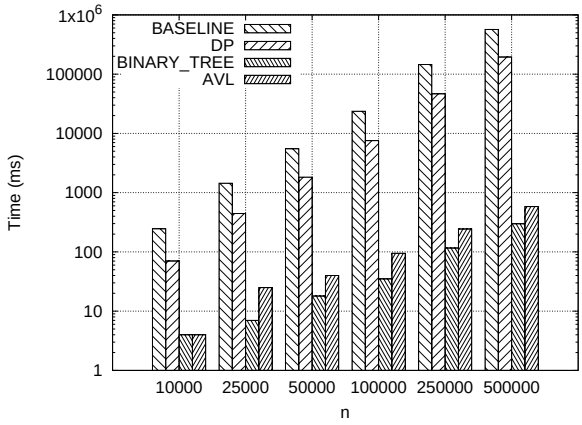


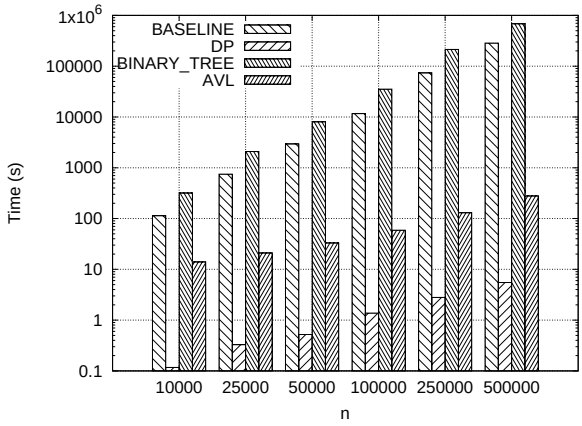
图 4: 后续排列优化算法性能分析

7.2 生成中介数的对比实验

我们依次令 n 取不同的值，以字典序为例，比较生成中介数的常规算法（BASELINE），动态规划算法（DP），二叉树算法（BINARY_TREE），平衡二叉树算法（AVL）对随机生成排列和均匀排列的运行时间。其中均匀排列即为 $\sum_{i=1}^{n-1} |I[i] - I[i+1]|$ 较小的排列，实验中取 $P = 1, 2, 3, \cdots, n$。



(a) 随机数据



(b) 均匀数据

图 5: 中介数优化算法性能分析

图 5 给出了利用常规算法和优化算法求中介数的实验结果。可以看到，对于随机数据，三种优化算法均提高了生成中介数的效率。从优化效果上来看，BINARY_TREE > AVL > DP。由于基于二叉树的中介数求解算法效率较原算法有量级提升（由 $O(n^2)$ 到 $O(n * log(n))$），故其效率远高于 DP 算法；而 DP 算法大致可以加速一般算法 3 ∼ 3.5 倍。由于数据随机，使得非平衡二叉树也基本处于平衡的状态，故

BINARY_TREE 算法整体上优于 AVL 算法。

对于均匀数据，可以看到 DP 算法是三种优化算法中最快的，由于数据较为均匀，故而算法时间复杂度接近 $O(n)$。而 BINARY_TREE 算法不仅没有加速常规算法，反而使得效率变低。这是由于此时二叉树处于极度不平衡状态，基本接近于线性查询，故而效率变低。但此时 AVL 算法和随机数据时性能基本保持一致。

综合以上分析，我们可以得出以下结论：1）AVL 算法在随机数据和均匀数据上都有较好的性能，表现相对稳定，可以应用于对算法稳定性要求较大的场景；2）若数据较为均匀，采用 DP 算法可以有更好的加速效果；反之，采用 BINARY_TREE 算法可以得到更好的结果。因此在求解中介数之前，可以通过判断 $\sum_{i=1}^{n-1} |I[i] - I[i+1]|$ 的值来决定采用何种算法。

8 总结与展望

本文深入研究了全排列的五种经典算法，普通字典序法，基于中介数的字典序法、递增进位制数法，递减进位制法和邻位对换法，并对比了其算法效率。同时，在这五种算法的基础上，结合动态规划和二叉树的思想，提出了三种优化求解中介数的算法。除此之外，对于求解当前排列后特殊位的排列，通过利用当中介数没有发生进位时的特殊性，我们省略了生成中介数的步骤，通过移动原排列数的若干位，直接生成新的排列。实验结果表明，我们的优化能够显著提高全排列算法的性能。同时，生成中介数的三种优化算法对分布特点不同的排列，还表现出不同的优化能力，因此我们需要根据应用场景选择合适的算法，以达到优化的目的。

References

[1] G. M Adel Son-Vel Skiĭ and E. M Landis. *An algorithm for the organization of information*. Libraries Unlimited, 1962, pp. 4671–4678.

[2] I. R Maccallum. “Surveyor’s Forum: A Forgotten Generation of Permutations”. In: *Acm Computing Surveys* 9.4 (1977), pp. 316–317.

[3] Robert Sedgewick. “Permutation Generation Methods”. In: *Acm Computing Surveys* 9.2 (1977), pp. 137–164.

[4] 卢开澄 and 卢华明. *组合数学. 第 4 版*. 清华大学出版社, 2006.

[5] 李模刚. “全排列生成算法在克莱姆法则中的应用”. In: *现代计算机: 专业版* 9 (2010), pp. 13–15.