

# 排列生成算法比较分析

## 组合数学大作业一

马晟

2009212623

2010-3-27

本文是组合数学第一次大作业说明文档，本次作业中，我使用 Java 作为编程语言，实现了两种排列生成算法，并在此说明文档中进行比较和分析。

目录

- 一、序数法 Ordinal.....3
  - 1. 算法说明.....3
  - 2. 程序实现.....3
- 二、字典序法.....4
  - 1. 算法说明.....4
  - 2. 程序实现.....4
- 三、换位法（利用队列改进算法） .....4
  - 1. 算法说明.....4
  - 2. 程序实现.....5
- 四、比较分析.....6
  - 1. 方法的比较分析.....6
  - 2. 生成全排列的时间比较.....6

# 一、序数法 Ordinal

## 1. 算法说明

序数法是利用了  $n! - 1 = (n-1)(n-1)! + (n-2)(n-2)! \cdots + 2 \times 2! + 1 \times 1!$  从而得到了一个从  $0 - (n! - 1)$  的数对的一一对应关系。对于  $n$  个数的全排列一共有  $n!$  个选择，每个选择可以得到一个一一对应的数组与之匹配，这样就可以通过这个数组求出每一个排列的形式。

在算法实现中，我实现了几个方法，可以实现三种表达形式的转换，即序数  $n \leftrightarrow$  数组  $a_n \leftrightarrow$  排列。

范例即书中的 1234 的全排列。

N	$a_3a_2a_1$	$p_1p_2p_3p_4$	N	$a_3a_2a_1$	$p_1p_2p_3p_4$
0	000	1234	12	200	1423
1	001	2134	13	201	2413
2	010	1324	14	210	1432
3	011	2314	15	211	2431
4	020	3124	16	220	3412
5	021	3214	17	221	3421
6	100	1243	18	300	4123
7	101	2143	19	301	4213
8	110	1342	20	310	4132
9	111	2341	21	311	4231
10	120	3142	22	320	4312
11	121	3241	23	321	4321

## 2. 程序实现

程序实现首先是要创建一个数列 **FNumber** 类，这个类表示数列  $(a_1, a_2, \dots, a_{n-1})$ ，程序默认  $n$  表示需要排列的个数，为了方便显示，排列的内容设为自然数  $1, 2, \dots, n$ 。（书中范例  $n=4$ ）。

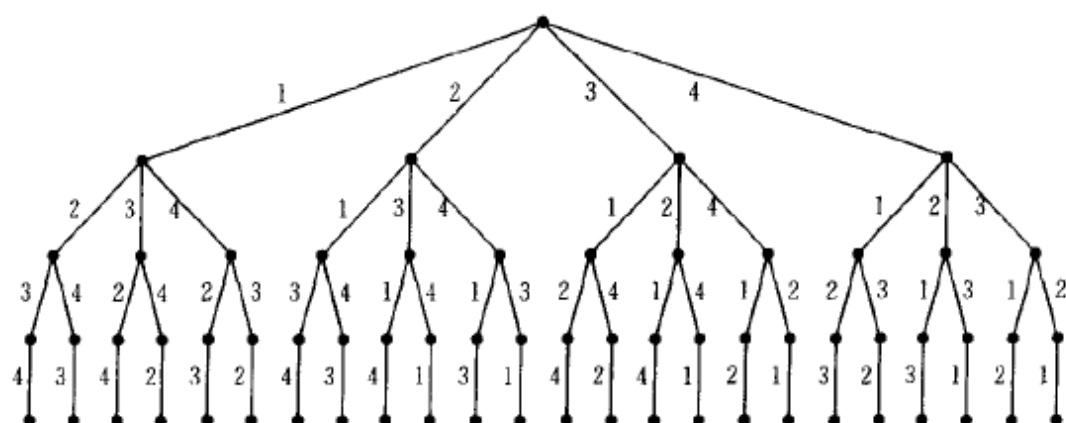
程序核心内容既是 **FNumber** 这个数的转换问题，从自然数  $N \rightarrow$  **FNumber** 的转换是类似于十进制到二进制的转换的方法得到的，**FNumber** 其实就是一个每位的进制都不一样的数字序列，个位是 2 进制，十位是 3 进制，百位是 4 进制……有书中方法可以得到转换。

从 **FNumber**  $\rightarrow$  排列的转换需要用到一个数组，对于每个  $a_i$  来说可以看成  $p$  中数  $i+1$  所在位置右边比  $i+1$  小的数的个数，这样可以在这个数组中先插入 1，然后根据  $a_i$  的值插入数  $i+1$ 。  
**Location** 这个变量即为数  $i+1$  的位置。

## 二、字典序法

### 1. 算法说明

字典序法主要是构建一棵树，每个节点没有取值，而节点间的路径为所选择的排列元素。如下图所示，选取的顺序为顺序选取，首先选 1，如果选择 1 之后，就继续按照 2, 3, 4 来选择分支。



字典序法示例图 (n=4)

对于查找一个排列的下一个排列的算法由书中所提示的方法，具体代码中有说明。

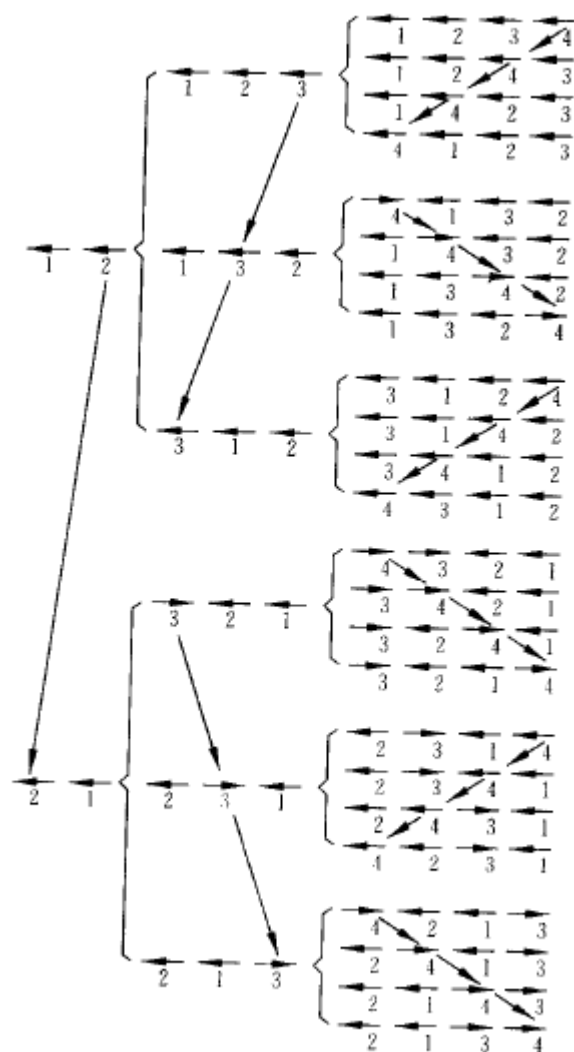
### 2. 程序实现

`getNext()`方法可以从一个排列得到下一个排列，如果该排列是最后一个，则得到这个排列本身。每一个排列是一个 `ArrayList`，由于可以从第一个排列根据循环使用排列得到全部排列，所以没有实现这个树状结构的数据结构，没有太大用途。

## 三、换位法（利用队列改进算法）

### 1. 算法说明

换位法的思想有点类似于轮转法，开始选定一个数，然后依次加入 2, 3, ..., n。得到一棵树，树的每个节点都是一个排列，设数的根节点为第一层，含有元素 1，第 i 层的所有叶子节点就是  $n=i$  的全排列。可以根据深度有限遍历的方法得到所有排列（这样得到的所有排列是有序的，即根据树的节点至上而下，如图所示）。



换位法图

## 2. 程序实现

构造一个树形结构，每个节点保存现有的排列元素串，以及一个状态指示，即为箭头（表示出它的子节点是以什么样的顺序插入的。），此外，每个节点也记录它的父亲节点，这样就可以从该节点找到它的下一个节点的排列情况。整个树的叶子节点即位所有排列。第  $i$  层所有叶子节点即位  $n=i$  的全排列。

每个节点都一个 `addSons()` 的方法可以得到所有的子节点。利用一个队列，将根节点记录在一个队列中。对此队列进行操作，每一次操作如下：首先判定队列第一个元素的排列长度是否为我们所求得的全排列长度，若是直接返回该队列，若不是，则对此节点调用 `addSons()` 的方法，并把得到的节点都加入到队列中，并把最开始的节点剔除队列。

循环这种操作，当停止时即得到了全排列。

## 四、比较分析

### 1. 方法的比较分析

换位法是从 1 个数的全排列开始，逐步地通过那个队列求得 2 个数的全排列，3 个数的全排列。如果存储空间足够的话，在应用换位法的时候，可以逐步求出  $n=1,2,3,\dots$  的全排列。

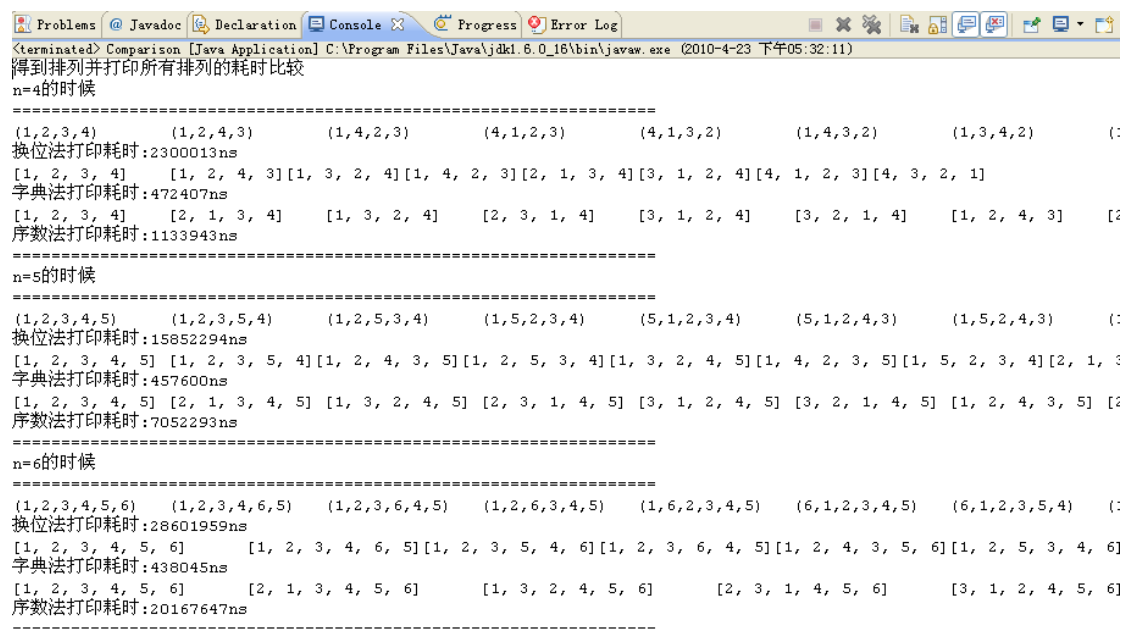
字典法更注重的是得到下一个排列，由字典法提供的法则，我已知一个排列可以很快地求出下一个排列是什么。而如果要得到所有排列，就可以如此从第一个意志得到最后一个。

序数法主要就是一个转换的过程，把一个排列用一个数列  $(a_1, a_2, \dots, a_{n-1})$  来表示，这样的活，给出任何一个位置，我都可以很快地求出这个位子对应的排列是什么。

总体上来说，如果我要得到各种  $n$  的全排列，使用换位法一次就可以得到全部的。如果我想得到一个可以快速查询的全排列，应用序数法会更好。如果我要得到一种链表式的全排列，那么应用字典法会比较好。字典法更像是另外两种方法的折中体现。关于代码的进一步研究可以参考 test 文件夹中的类，这些都是针对我的代码写的一些测试，并不是严格遵照 test case 的方式书写的，只是用来测试代码的正确性。

### 2. 生成全排列的时间比较

如图 4-1，换位法生成全排列的时间会长许多，而字典法是相对很短的。在 `comprison` 的包中，我提供了一个比较的测试方法，可以调用得到结果。三种方法的主类中都包含有 `printAllPermutation()` 的方法用于比较，这个方法的实现也可以体现出如何应用各个类的方法得到全排列。



```
<terminated> Comparison [Java Application] C:\Program Files\Java\jdk1.6.0_16\bin\javaw.exe (2010-4-23 下午05:32:11)
得到排列并打印所有排列的耗时比较
n=4的时候
=====
(1,2,3,4)      (1,2,4,3)      (1,4,2,3)      (4,1,2,3)      (4,1,3,2)      (1,4,3,2)      (1,3,4,2)      (:
换位法打印耗时:2300013ns
[1, 2, 3, 4] [1, 2, 4, 3] [1, 3, 2, 4] [1, 4, 2, 3] [2, 1, 3, 4] [3, 1, 2, 4] [4, 1, 2, 3] [4, 3, 2, 1]
字典法打印耗时:472407ns
[1, 2, 3, 4] [2, 1, 3, 4] [1, 3, 2, 4] [2, 3, 1, 4] [3, 1, 2, 4] [3, 2, 1, 4] [1, 2, 4, 3] [2, 1, 4, 3]
序数法打印耗时:1133943ns
=====
n=5的时候
=====
(1,2,3,4,5)      (1,2,3,5,4)      (1,2,5,3,4)      (1,5,2,3,4)      (5,1,2,3,4)      (5,1,2,4,3)      (1,5,2,4,3)      (:
换位法打印耗时:15852294ns
[1, 2, 3, 4, 5] [1, 2, 3, 5, 4] [1, 2, 4, 3, 5] [1, 2, 5, 3, 4] [1, 3, 2, 4, 5] [1, 4, 2, 3, 5] [1, 5, 2, 3, 4] [2, 1, 3, 4, 5]
字典法打印耗时:457600ns
[1, 2, 3, 4, 5] [2, 1, 3, 4, 5] [1, 3, 2, 4, 5] [2, 3, 1, 4, 5] [3, 1, 2, 4, 5] [3, 2, 1, 4, 5] [1, 2, 4, 3, 5] [2, 1, 4, 3, 5]
序数法打印耗时:7052293ns
=====
n=6的时候
=====
(1,2,3,4,5,6)      (1,2,3,4,6,5)      (1,2,3,6,4,5)      (1,2,6,3,4,5)      (1,6,2,3,4,5)      (6,1,2,3,4,5)      (6,1,2,3,5,4)      (:
换位法打印耗时:28601959ns
[1, 2, 3, 4, 5, 6] [1, 2, 3, 4, 6, 5] [1, 2, 3, 5, 4, 6] [1, 2, 3, 6, 4, 5] [1, 2, 4, 3, 5, 6] [1, 2, 5, 3, 4, 6]
字典法打印耗时:438045ns
[1, 2, 3, 4, 5, 6] [2, 1, 3, 4, 5, 6] [1, 3, 2, 4, 5, 6] [2, 3, 1, 4, 5, 6] [3, 1, 2, 4, 5, 6]
序数法打印耗时:20167647ns
=====
```

图 4-1  $n=4,5,6$  时生成全排列的时间比较