

# Discrete

## Mathematics - P153

将可区分的物体放入不可区分的盒子，且每个盒子不为空

第二类斯特林数 (Stirling number of the second kind)

对于正整数  $n \geq k$ , 将  $n$  个可区分的物体放入  $k$  个不可区分的盒子，且盒子均不为空的方式数

记为  $\{n\}_k$ , 或  $S_{n,k}$ , 或  $S_n^{(k)}$ , 称为第二类斯特林数

$$\sum_{i=0}^{k-1} (-1)^i \frac{1}{k!} \cdot \binom{k}{i} \cdot (k-i)^n$$

$$S_{n,k} = \boxed{\text{计算公式}} \quad \text{其中正整数 } n \geq k$$

证明过程有：首先对  $k$  个盒子进行编号  $1, 2, \dots, k$ ,

则令情况  $C_i$  表示某中分配中，第  $i$  个盒子为空。

可知将  $n$  个可区分的物体放入  $k$  个不可区分的盒子，且盒子均不为空的方式数

可表示为  $N(\bar{C}_1, \bar{C}_2, \dots, \bar{C}_k)$  至少有

令  $S_i$  表示对  $C_1, C_2, \dots, C_k$  中  $i$  个情况发生的方式数

则根据容斥原理有

$$N(\bar{C}_1, \bar{C}_2, \dots, \bar{C}_k) = S_0 - S_1 + S_2 - \dots + (-1)^{k-1} S_{k-1}$$

对于  $S_i$  可做如下考虑，其中  $i=0, 1, 2, \dots, k-1$

1. 从  $k$  个盒子中选择  $i$  个盒子定为空，则有  $\binom{k}{i}$  种方式

2. 将  $n$  个物体任意地放入剩余  $k-i$  个盒子，则有  $(k-i)^n$  种方式

3. 由于  $k$  个盒子不可区分，所以方式数应除以  $k!$

$$\text{于是有 } S_i = \binom{k}{i} (k-i)^n / k!$$

$$\begin{aligned} \text{即 } N(\bar{C}_1, \bar{C}_2, \dots, \bar{C}_k) &= S_0 - S_1 + S_2 - \dots + (-1)^{k-1} S_{k-1} \\ &= \sum_{i=0}^{k-1} (-1)^i S_i \end{aligned}$$

$$\text{于是有 } S_{n,k} = \sum_{i=0}^{k-1} (-1)^i \binom{k}{i} (k-i)^n / k!$$

对于正整数  $n \geq k \geq 1$ , 有  $S_{n,k} = S_{n-1,k-1} + k \cdot S_{n-1,k}$  且盒子均非空

组合证明过程为：对于将  $n$  个可区分的物体放入  $k$  个不可区分的盒子，其中  $n \geq k \geq 1$

考虑物体 1 的归属情况，有两种可能

1. 物体 1 单独在某个盒子中，则等价于  $S_{n-1,k-1}$

即将剩余  $n-1$  个物体放入  $k-1$  个盒子且盒子均非空

2. 物体 1 与其他元素在同一个盒子中，则等价于  $k \cdot S_{n-1,k}$

即先将剩余  $n-1$  个物体放入  $k$  个盒子中，有  $S_{n-1,k}$  种方式

再将物体 1 放入  $k$  个盒子中的一个

由于此时  $k$  个盒子已放入物体，所以可看作可区分的盒子，即  $\binom{k}{1}$  种方式

即有  $k \cdot S_{n-1,k}$  种方式

$$\text{于是有 } S_{n,k} = S_{n-1,k-1} + k \cdot S_{n-1,k}, \text{ 其中 } n \geq k \geq 1$$

# Discrete

## Mathematics - P154

对于正整数  $n < k$ , 第二类斯特林数  $S(n, k) = 0$   
 证明过程有, 将  $n$  个可区分的物体放入  $k$  个不可区分的盒子  
 而如果  $n < k$ , 则无法实现  $k$  个盒子均非空, 即  $S(n, k) = 0$

对于正整数  $n, k$ , 当  $k=1$  或  $n=k$  时,  $S(n, k) = 1$   
 证明过程有, 如果仅有 1 个盒子, 则放入  $n$  个物体的方式仅有 1 种  
 如果有恰好  $n$  个盒子, 则放入  $n$  个物体且盒子均非空的方式仅有 1 种

对于正整数  $n, k$ , 当  $k=n-1$  时, 有  $S(n, k) = \binom{n}{2}$   
 证明过程有, 将  $n$  个可区分的物体放入  $n-1$  个不可区分的盒子, 且盒子均非空  
 则有且仅有 2 个物体被放入 1 个盒子, ~~且~~ 有  $\binom{n}{2}$  种选择方式  
 而其余  $n-2$  个物体放入  $n-2$  个盒子, 仅有 1 种方式  
 于是有  $\binom{n}{2}$  种方式, 即  $S(n, n-1) = \binom{n}{2}$

对于正整数  $n, k$ , 当  $k=2$  时, 有  $S(n, k) = 2^{n-1} - 1$   
 证明过程有, 将  $n$  个可区分的物体放入 2 个不可区分的盒子, 且 ~~2~~ 盒子均非空  
~~也可以描述为在  $n$  个元素的集合上的等价关系~~

不同的有两个等价类的等价关系的关系数

$$\text{则有 } S(n, 2) = [\sum_{k=1}^{n-1} \binom{n}{k}] / 2 = [\sum_{k=0}^n \binom{n}{k} - \binom{n}{0} - \binom{n}{n}] / 2 = 2^{n-1} - 1$$

$S(n, k)$	$k=1$	$2$	$3$	$\dots$	$k$	$\dots$	$n-1$	$n$
$n=1$	1	0	0	$\dots$	-	$\dots$	-	1
2	1	1	0	$\dots$	-	$\dots$	1	1
3	1	3	1	$\dots$	-	$\dots$	3	1
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	-	$\dots$	$\dots$	$\dots$
$n$	1	<del><math>2^{n-1}</math></del>	-	$\dots$	$\frac{1}{k!} \sum_{i=0}^{k-1} (-1)^i \binom{k}{i} (k-i)^n$	$\dots$	$\binom{n}{2}$	1

问: 递归地定义函数  $S(n, k)$ , 其中  $n, k \in \mathbb{Z}^+$

$$S(n, k) = \begin{cases} 0 & , n < k \\ 1 & , k=1 \\ \dots & , k=n \\ S(n-1, k-1) + k \cdot S(n-1, k) & , 1 < k < n \end{cases}$$

secondStirling n k // secondStirling :: Int -> Int -> Int

$$\begin{cases} n < k & \Rightarrow 0 \\ k=1 & \Rightarrow 1 \\ k=n & \Rightarrow 1 \\ \text{Otherwise} & = \text{secondStirling}(n-1, k-1) + k * \text{secondStirling}(n-1, k) \end{cases}$$

# Discrete

## Mathematics - P155

将可区分的物体放入不可区分的盒子

对于将  $n$  个可区分的物体放入  $k$  个不可区分的盒子，其中  $n, k \in \mathbb{Z}^+$

有  $\sum_{j=1}^k \frac{1}{j!} \sum_{i=0}^{j-1} (-1)^i \binom{j}{i} (j-i)^n$  种不同方式

证明过程有，将  $n$  个可区分的物体放入  $k$  个不可区分的盒子，  
由于允许盒子为空，则可按非空的盒子数来划分情况

对于正整数  $1 \leq j \leq k$ ，假设非空的盒子有了  $j$  个

又盒子不可区分，则只有 1 种方式从  $k$  个盒子中选择  $j$  个

于是转换为  $n$  个物体放入  $j$  个盒子且盒子非空的方式数

即有  $S(n, j)$  种不同方式

现合计有  $\sum_{j=1}^k S(n, j)$  种方式

即  $\sum_{j=1}^k S(n, j) = \sum_{j=1}^k \frac{1}{j!} \sum_{i=0}^{j-1} (-1)^i \binom{j}{i} (j-i)^n$

### 贝尓數

(Bell number), 记为  $B_n$ ，其中  $n$  为非负整数

表示对于包含  $n$  个元素的集合的不同划分的方式数，或定义在其上的不同等价关系的个数

different ways to partition a set containing exactly  $n$  element

or equivalently, number of equivalence relations on the set

集合  $S$  的划分 (partition) 定义为一个集合，

其元素为集合  $S$  的一组非空子集，且两两交集为空，且全部并集为集合  $S$

nonempty, pairwise disjoint subsets of  $S$  whose union is  $S$

特别地有,  $B_0 = 1$ , 由于空集的划分只有空集

且空集的成员都是非空集合是空证明 (vacuously true), 且所有成员并集为空集

贝尓數有递归关系,  $B_{n+1} = \sum_{k=0}^n \binom{n}{k} B_k$  其中  $n \in \mathbb{N}$

证明过程有, 对于  $n+1$  个元素的划分数  $B_{n+1}$ , 考虑划分中第  $n+1$  个元素所处的子集

假设该集合中除  $a_{n+1}$  外有  $k$  个元素, 其中  $0 \leq k \leq n$

则可描述为从前  $n$  个元素中选取  $k$  个, 与  $a_{n+1}$  放入同一个子集, 则有  $\binom{n}{k}$  种划分

再对剩余  $n-k$  个元素进行划分, 则有  $B_{n-k}$

于是对于任意  $0 \leq k \leq n$ , 有  $\binom{n}{k} B_{n-k}$  种方式

现合计有  $B_{n+1} = \sum_{k=0}^{n+1} \binom{n}{k} B_{n-k} = \sum_{k=0}^n \binom{n}{n-k} B_{n-k} = \sum_{t=0}^n \binom{n}{t} B_t$ , 其中  $t = n-k$

贝尓數与第二类斯特林数关系为,  $B_n = \sum_{k=0}^n S(n, k)$ ,  $n \in \mathbb{Z}^+$

证明过程有,  $B_n$  可描述为将  $n$  个物体放入  $n$  个不可区分的盒子,

则相当于选择  $1 \leq k \leq n$  个盒子非空, 则有  $S(n, k)$  种方式放入  $n$  个物体.

于是有  $B_n = \sum_{k=1}^n S(n, k) = \sum_{k=1}^n \frac{1}{k!} \sum_{i=0}^k (-1)^i \binom{k}{i} (k-i)^n$

# Discrete

## Mathematics - P156

例 9 - multiplication

贝尔数可以递归地定义。在非负整数上，即可以递归地定义  $B_n$ ,  $n \in \mathbb{N}$

$$\text{有 } B_n = \begin{cases} 1 & n=0 \\ \sum_{k=0}^{n-1} \binom{n}{k} B_k, & n>0 \end{cases}$$

则 bell Number :  $\text{Int} \rightarrow \text{Int}$

bell Number 0 = 1 基础步骤  $B_0 = 1$

bell Number  $n =$  第  $n+1$  次迭代前, lst 第一个元素即为  $B_n$ .

let func o [lst] = head lst 当  $n$  次迭代完成后

func k lst =  $\lambda$  = 将 lst 转换为  $\lambda$  [(k-t,  $B_t$ )] 的形式

let enum = myEnumerate lst 在第  $k$  次迭代中计算  $\binom{n-k}{t} B_t$

func (ind, ele) => acc + (myCombination (n-k) ind) \* ele

$\frac{d}{dx} + \frac{d}{dx} = \frac{1}{x!} \cdot x! + x = \text{foldl } f o \text{ enum }$  计算  $B_{n-k+1} = \sum_{t=0}^{n-k} \binom{n-k}{t} B_t$

$\lambda$  [in func (k-1) (x: lst)] 保存  $B_{n-k+1}$  到列表第一个元素。

$\lambda$  [in func n [1]] 并进入下一次迭代

注意，由于  $B_n$  的递归调用会调用  $B_0, B_1, \dots, B_{n-1}$

所以直接使用递归定义函数，可能导致复杂度大幅上升

于是改用循环形式，在有  $n$  次迭代的循环中

第  $k$  次迭代前，列表 lst 的第一个元素记录了  $B_{k-1}$  的值

多项式定理 (multinomial theorem), 即对二项式定理推广到有  $t$  个项的情形 且,  $0 \leq n_1, n_2, \dots, n_t \leq n$   
对于非负整数  $n$ ,  $(x_1 + x_2 + \dots + x_t)^n = \sum \frac{n!}{n_1! n_2! \dots n_t!} x_1^{n_1} x_2^{n_2} \dots x_t^{n_t}$ , 其中  $n_1 + n_2 + \dots + n_t = n$

归纳法证明过程，基础步骤为二项式定理。即  $(x_1 + x_2)^n = \sum \frac{n!}{n_1! n_2!} x_1^{n_1} x_2^{n_2}$

递归步骤：假设对于任意  $t \geq 2$ , 有  $P(t)$  为真，则考虑  $P(t+1)$

$$(x_1 + x_2 + \dots + x_{t+1})^n = [(x_1 + x_2 + \dots + x_t) + x_{t+1}]^n$$

$$= \sum_{n_{t+1}=0}^n x_{t+1}^{n_{t+1}} (x_1 + x_2 + \dots + x_t)^{n-n_{t+1}} \cdot \binom{n}{n_{t+1}}$$

$$\text{其中 } n_1 + n_2 + \dots + n_t = n - n_{t+1}, 0 = (t+1) \stackrel{(IH)}{=} \sum_{n_{t+1}=0}^n x_{t+1}^{n_{t+1}} \sum_{n_1, n_2, \dots, n_t} \frac{(n-n_{t+1})!}{n_1! n_2! \dots n_t!} \cdot x_1^{n_1} x_2^{n_2} \dots x_t^{n_t} \cdot \frac{n!}{(n-n_{t+1})! n_{t+1}!}$$

$$\text{且 } 0 \leq n_1, n_2, \dots, n_t \leq n - n_{t+1}, 0 = (t+1) \stackrel{(IH)}{=} \sum_{n_{t+1}=0}^n \frac{n!}{n_1! n_2! \dots n_t! n_{t+1}!} x_1^{n_1} x_2^{n_2} \dots x_t^{n_t} x_{t+1}^{n_{t+1}}$$

$$= \sum_{n_{t+1}=0}^n \frac{n!}{n_1! n_2! \dots n_t! n_{t+1}!} x_1^{n_1} x_2^{n_2} \dots x_t^{n_t} x_{t+1}^{n_{t+1}}, \text{ 即 } P(t+1) \text{ 为真}$$

于是根据数学归纳法，对于任意非负整数  $n, t \geq 2$

$$(x_1 + x_2 + \dots + x_t)^n = \sum \frac{n!}{n_1! n_2! \dots n_t!} x_1^{n_1} x_2^{n_2} \dots x_t^{n_t}, \text{ 其中 } n_1 + n_2 + \dots + n_t = n, 0 \leq n_1, n_2, \dots, n_t \leq n$$

组合法证明过程有，对于  $(x_1 + x_2 + \dots + x_t)^n$  展开式中的某一项  $x_1^{n_1} x_2^{n_2} \dots x_t^{n_t}$

有  $n_1 + n_2 + \dots + n_t = n$ , 且  $0 \leq n_1, n_2, \dots, n_t \leq n$ .

则可以描述为将  $n$  个物体放入  $t$  个盒子，使得盒子  $i$  中包含  $n_i$  个物体的方式数之

$$\text{于是有 } \binom{n}{n_1} \binom{n-n_1}{n_2} \dots \binom{n-n_{t-1}}{n_t} = \frac{n!}{n_1! n_2! \dots n_t!}$$

$$\text{所以有 } (x_1 + x_2 + \dots + x_t)^n = \sum \frac{n!}{n_1! n_2! \dots n_t!} x_1^{n_1} x_2^{n_2} \dots x_t^{n_t}, \text{ 其中 } n_1 + n_2 + \dots + n_t = n, 0 \leq n_1, n_2, \dots, n_t \leq n$$

# Discrete

## Mathematics - P157

P7 - Selection

生成排列

对于由  $\{1, 2, \dots, n\}$  构成的排列  $a_1, a_2, \dots, a_n$  可以按字典序生成下一个排列。

可以按字典序生成大于  $a_1, a_2, \dots, a_n$  的下一个排列。

procedure NEXT\_PERMUTATION( $\{a_1, a_2, \dots, a_n\}$ )

$j := n - 1$

while  $a_j > a_{j+1}$ :

$j := j - 1$

$k := n$

while  $a_j > a_k$ :

$k := k - 1$

swap( $a_j, a_k$ )

$r := n$

$s := j + 1$

while  $r > s$ :

swap( $a_s, a_r$ )

$r := r - 1$

$s := s + 1$

这一部分是找到一个下标  $j$ ,

使得  $a_{j+1}, a_{j+2}, \dots, a_n$  为一个降序的子序列

且  $a_j < a_{j+1}$

这一部分是找到一个下标  $j+1 \leq k \leq n$ .

使得  $a_k$  为  $a_{j+1}, \dots, a_n$  中大于  $a_j$  的元素中最大的一个

由于  $a_{j+1}, \dots, a_n$  为降序, 所以第一个使  $a_j < a_k$  的即为

swap( $a_j, a_k$ )

交换  $a_j$  与  $a_k$ ,

这一部分是将新的  $a_{j+1}, \dots, a_n$  整理为升序

由于对于原有的  $a_k$ , 有  $a_{k-1} > a_k > a_{k+1}$

又  $a_k > a_j > a_{k+1}$

于是在交换后仍旧满足  $a_{k-1} > a_k > a_{k+1}$

即  $a_{j+1}, \dots, a_n$  仍旧保持降序

则依次交换首尾的元素即可得到  $a_{j+1}, \dots, a_n$  为升序

{此时后的  $a_1, a_2, \dots, a_n$  即为下一个排列}

在 Python 中的实现为

```
def next_permutation(lst):
```

$n = len(lst)$

$j = n - 2$

注意 Python 中  $lst$  下标为  $[0, n-1]$

while  $lst[j] > lst[j+1]$ :

$j -= 1$

$k = n - 1$

while  $lst[j] > lst[k]$ :

$k -= 1$

$lst[j], lst[k] = lst[k], lst[j]$

$s, r = j + 1, n - 1$

while  $r > s$ :

$lst[s], lst[r] = lst[r], lst[s]$

$s, r = s + 1, r - 1$

注意由于 Python 中的多变量赋值特性

$swap(a, b)$  不必单独实现为函数

而是用  $a, b = b, a$  的赋值语句

注意在 Python 中对于传入参数都是采用传引用的方式

所以在函数中对可变对象  $list$  的改变会影响原变量

# Discrete

## Mathematics - P158

生成排列

对于给定的初始列表，通过 swap 方法生成列表的全排列

swap 方法可以交换两个元素在 list 中的位置

基本思路是，对于长度为 n 的列表，标记第 n 个元素

依次使第 n 个元素与第 1 个, ..., 第 n 个元素交换

再递归地调用长度为 n-1 的列表的全排列

`def Permute([ ] lst, n):` 生成长度为 n 的列表 lst 的全排列

`if n == 1:` [ ] 基础情形，当 n=1 时，打印当前序列

`print(lst)`

`else:`

`for k in range(n):` [ ] 递归情形，for k=0 to n-1

向右操作

$\rightarrow \text{lst}[k], \text{lst}[n-1] = \text{lst}[n-1], \text{lst}[k]$  swap( $\text{lst}[k], \text{lst}[n-1]$ )

回到初始排列

递归地调用  $\text{Permute}(\text{lst}, n-1)$

$\text{lst}[k], \text{lst}[n-1] = \text{lst}[n-1], \text{lst}[k]$  swap( $\text{lst}[k], \text{lst}[n-1]$ )

可以通过将树的结点描述为对函数的调用，将边描述为 swap 操作

将函数生成全排列的过程描述为递归树，并计算由 swap 操作计量的复杂度

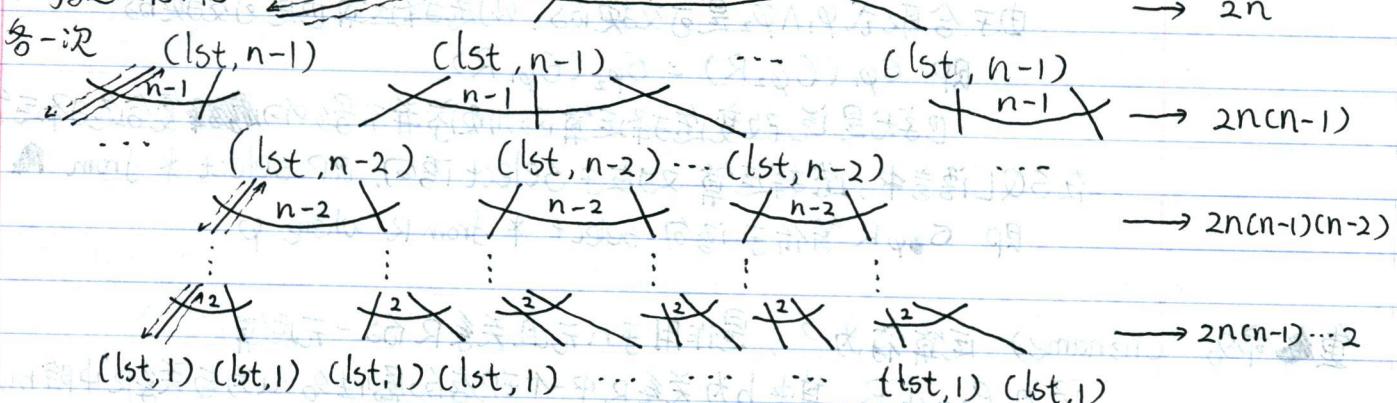
注意在每个 iteration 中有两次 swap 操作。

即在遍历过程中，

每条边上行下行

各一次

`permute(lst, n)`



$$\text{于是可知 } T(n) = 2n + 2n(n-1) + \dots + 2n(n-1)\dots 2$$

$$= 2P(n, 1) + 2P(n, 2) + \dots + 2P(n, n-1)$$

$$= 2 \frac{n!}{(n-1)!} + 2 \frac{n!}{(n-2)!} + \dots + 2 \frac{n!}{1!}$$

$$= 2n! \left( 1 + \frac{1}{2!} + \dots + \frac{1}{(n-1)!} \right) = 2n! \left( \sum_{k=0}^{n-1} \frac{1}{k!} - \frac{1}{0!} \right)$$

又当  $n \rightarrow +\infty$  时， $\sum_{k=0}^{n-1} \frac{1}{k!} \rightarrow e$

于是有  $T(n) \sim 2(n-1)n!$

# Discrete

## Mathematics - P159

生成排列

对于基于 swap 方法的排列生成算法，如果以 swap 的次数计算复杂度

注意在函数循环的每次 iteration 中，使用了 2 次 swap。

从递归树的角度考虑，相当于沿着每条边移动了 2 次。

则考虑将每个 iteration 中的 swap 压缩到 1 次。

对于  $n$  为奇数的情形，始终交换  $A[1]$  与  $A[n]$ 。

对于  $n$  为偶数的情形，在第  $k$  次迭代交换  $A[k]$  与  $A[n]$ 。

```
def myPermute(lst, n):
    if n == 1:
        print(lst)
    else:
        for k in range(1, n+1):
            lst[0], lst[n-1] = lst[n-1], lst[0]
            myPermute(lst, n-1)
            if n % 2 == 1:
```

归内而言：

当  $n$  为奇数的函数调用返回

如果 ~~■~~ 初始序列为

$a_1, a_2, \dots, a_n$

则保持不变

当  $n$  为偶数的函数调用返回

如果初始序列为

$a_1, a_2, \dots, a_n$

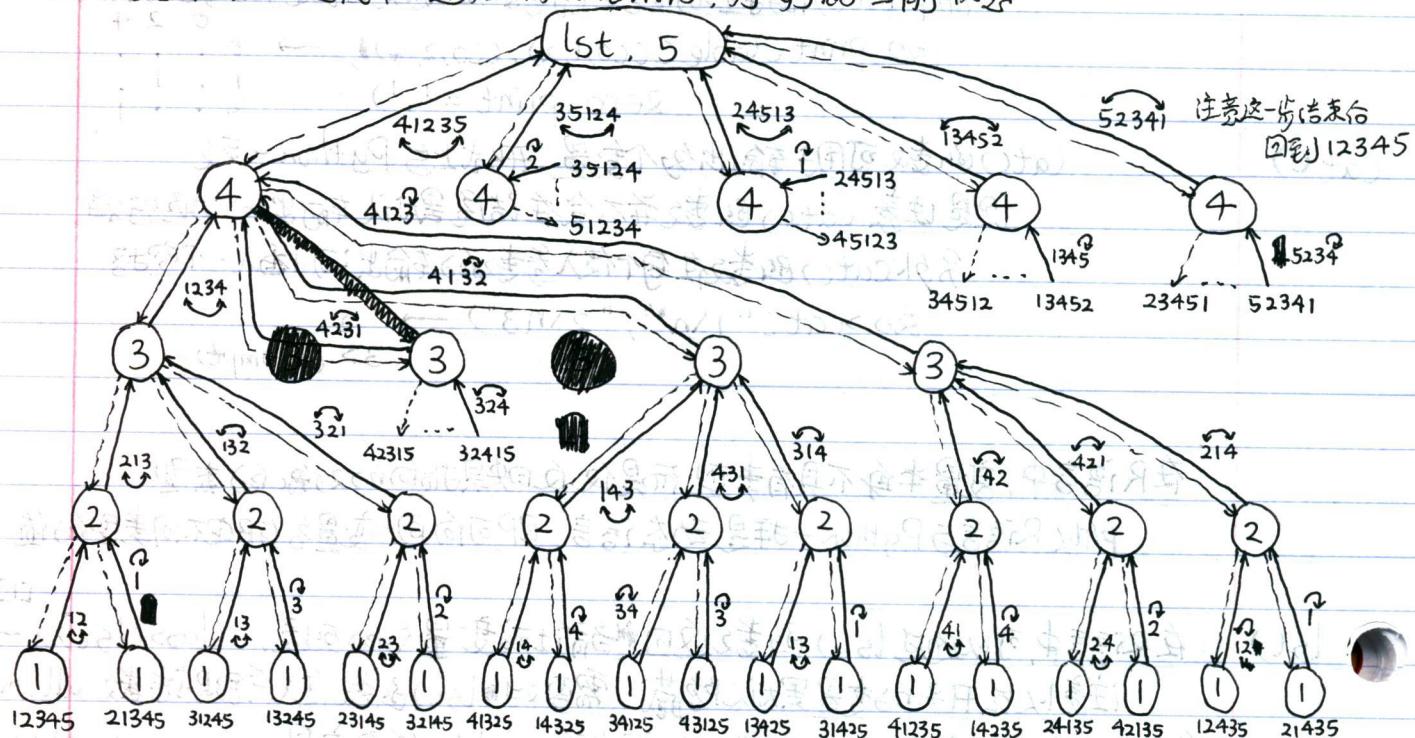
则返回时整体向右移动

即有  $a_n, a_1, a_2, \dots, a_{n-1}$

$lst[0], lst[n-1] = lst[n-1], lst[0]$

$lst[k-1], lst[n-1] = lst[n-1], lst[k-1]$

考虑在某次迭代中，递归调用完成后，序列的当前状态



注意，由于在递归树中，实际上只有父子节点，返回上一层时有 swap 操作，所以每条边只计算 1 次。

$$\text{即有复杂度 } T(n) = \sum_{k=1}^{n-1} \frac{n!}{k!} \sim (e-1)n!$$

# Discrete

Mathematics - P160 [www.mathsrevision.net](http://www.mathsrevision.net)

生成排列 对于以 swap 方法为基础的排列生成算法，以 swap 操作次数为基准评价复杂度  
注意递归的方法当前的复杂度为  $T(n) \approx (e-1)n! > n!$

而即使  $n$  为偶数时的最后一次 iteration 的 swap 操作省去

其复杂度依旧大于  $n!$ 。于是考虑可以达到理论下限  $T(n) = n! - 1$  的算法。

以一个 pair (element, direction) 表示列表中的每个元素。

如  $[(-1, 1), (2, 1), (3, -1)]$  表示  $\vec{1}, \vec{2}, \vec{3}$  的一个子集

如果对某个元素，其方向上有一个元素，且该元素小于当前元素，则称其为可移动的。每一轮选择当前可移动的值最大的元素。

不断与其移动方向上的下一个元素交换位置，直到无法移动为止

此时选择下一个可移动的 最大元素

并将其值大于本轮转动元素的向后元素的方向反转

在每次移动后输出当前序列

当选中元素为列表中最小元素时，算法终止

初始化	↑1	↑1	↑1	↑4↑4	↓4	↑3↑3	↑3	↓3	↓3	↓3↑4↑4↓4	↑2	↑2↑2	1	1	1	2	2	2
↑2	↑2	↑4	↑1↑1	↑3↓4↑1	↑1	↑2	↑2	↑4	↓3↑2↑2	↓4↑1↑1	4	3	3	4	3	3		
↑3	↑4	↑1	↑2	↑2↑3	↑1	↑1	↓4	↑2	↑1	↑4	↑2	↑2	↓3↑1	↑1	↓4	↓3	3	4
↑4	↑3	↑3	↑3↑2	↓2	↑2	↑2	↑2	↓4	↑4↑4↑1	↑1	↑1	↑1	↓3	↓3	↓4	2	2	4

注意：在这个算法中，缺失了部分排列

其产生的原因在于缺失了包含 132 和 231 的排列

其产生的步骤聚为每一车包的终止时间

即应该在每一次移动后执行一次转向操作

并立刻重新选择可移动的元素

修改循环结构为

输出当前排列

## 循环结构

将当前最大的可移动元素向其移动方向移动一格

将比该元素的值大的元素的移动方向反转

重新选择最大的可移动元素

而当选中的元素为列表中最小元素时，循环终止

初始化  $\uparrow 1 \uparrow 1 \uparrow 1 \uparrow 4 \downarrow 4 \uparrow 1 \uparrow 1 \uparrow 3 \uparrow 3 \uparrow 3 \uparrow 4 \downarrow 4 \downarrow 3 \downarrow 3 \downarrow 3 \uparrow 2 \uparrow 2 \uparrow 2 \uparrow 4 \downarrow 4 \uparrow 2 \uparrow 2 \uparrow 2 \uparrow 2$

$\uparrow 2 \uparrow 2 \uparrow 4 \uparrow 1 \uparrow 1 \downarrow 4 \uparrow 3 \uparrow 3 \uparrow 1 \uparrow 1 \uparrow 1 \uparrow 4 \uparrow 3 \downarrow 3 \downarrow 4 \uparrow 2 \uparrow 2 \downarrow 3 \downarrow 3 \uparrow 4 \uparrow 2 \uparrow 2 \downarrow 4 \uparrow 1 \uparrow 1$

↑3 ↑4 ↑2 ↑2 ↑3 ↑3 ↓4 ↑2 ↑2 ↑4 ↑1 ↑1 ↑1 ↑2 ↑2 ↓4 ↑1 ↑1 ↑1 ↑4 ↓3 ↓3 ↑1 ↑1 ↓4 ↓3

↑4 ↑3 ↑3 ↑3 ↓2 ↑2 ↑2 ↓4 ↑4 ↑2 ↑2 ↑2 ↑1 ↑1 ↑1 ↓4 ↑4 ↑1 ↑1 ↑1 ↓3 ↓3 ↓3 ↓4

注意此时实际上没有可移动元素，而如果使 1 移动一步，则回到最初始排列

# Discrete

## Mathematics - P161

生成排列

基于 swap 方法 并使 swap 操作次数优化至  $n!-1$  次的 move 算法在 Python 中的实现

设计成 class 的形式，输入列表并初始化后，通过成员函数打印所有排列

```
class movePermute(object):
    -slots_ = ('lst', 'min')
    movePermute 类中只有两个属性,
    lst 用于记录元素和方向, min 记录最小元素

    def __init__(self, lst):
        lst.sort()
        self.lst = [[x, -1] for x in lst]
        self.min = self.lst[0][0]

    def __repr__(self):
        return str(self.lst)

    def __print__(self):
        print([ele for ele, _ in self.lst])  # 忽略方向

    def __turning__(self, x):
        for ele in self.lst:
            if ele[0] > x:
                ele[1] = 0 - ele[1]

    def __movable__(self, ind):
        next_ind = ind + self.lst[ind][1]
        return next_ind >= 0 and next_ind < len(self.lst) and self.lst[ind][0] > self.lst[next_ind][0]

    def __choose__(self):
        max, pick = self.min, -1
        for ind, ele in enumerate(self.lst):
            if ele[0] > max and self.__movable__(ind):
                max, pick = ele[0], ind
        return pick

    def __permute__(self):
        self.__print__()
        ind = self.__choose__()

        while ind >= 0:
            next_ind = ind + self.lst[ind][1]
            self.lst[ind], self.lst[next_ind] = self.lst[next_ind], self.lst[ind]
            self.__turning__(self.lst[next_ind][0])
            self.__print__()
            ind = self.__choose__()
```

注意：

虽然从 swap 的角度

确实优化到  $n!-1$

但由于选择和转向

实际上可能慢于递归算法

当可移动的最大元素不是最小元素时

交换该元素与其方向上

的下一个元素

并将其大于该元素的元素转向

选择下一个可移动的最大元素

# Discrete

## Mathematics - P162

生成下一个位串 对于给定的位串  $b_{n-1}, b_{n-2} \dots b_1, b_0$ , 轮出下一个位串

procedure NEXT-BIT-STRING (位串  $b_{n-1}, b_{n-2} \dots b_1, b_0$ , 且不等于11...11)

$i := 0$  遍历这个位串，直到遇到第一个值为1为止

while  $b_i = 1$  ] 对于结尾的所有连续的1，向右遍历

$b_i := 0$  为重置，将其置为0，继续向右遍历

$i := i + 1$

else  $b_i := 1$  [ 对于最右的一位0，置为1，继续向右遍历

return  $b_{n-1}, b_{n-2} \dots b_1, b_0$

] 按字典序生成下一个组合，对于给定的集合  $\{1, 2, \dots, n\}$  的  $r$  组合。

procedure NEXT-R-COMBINATION ( $a_1, a_2, \dots, a_r \in \{1, 2, \dots, n\}$ )

procedure NEXT-R-COMBINATION ( $a_1, a_2, \dots, a_r \in \{1, 2, \dots, n\}$ )

$i := r$

while  $a_i = n-r+i$  ] 找到最右的一个  $a_i$

$i := i - 1$  ] 使得  $a_i \neq n-r+i$

$a_i := a_i + 1$  ] 将该  $a_i + 1$  作为新的  $a_i$

for  $j := i+1 \dots r$  ] 对于  $a_{i+1}, \dots, a_r$ ，更新为  $a_i+1, a_i+2, \dots, a_i-i+r$

$a_j := a_i + j - i$  ] 即有  $\{a_1, a_2, \dots, a_i, a_i+1, \dots, a_i-i+r\}$

return  $\{a_1, a_2, \dots, a_r\}$

在 Python 中以生成器的方式实现以字典序输出  $n$  个元素集合的  $r$  组合

def combination\_gen(alpha, r): ] 以长度为  $n$  的字母表生成  $r$  组合

len\_alpha = len(alpha) ] 计算字母表的长度

lst = alpha[0:r] ] 初始化最小的  $r$  组合并返回

yield lst

while lst[0] != alpha[n-r]: ] 当前  $r$  组合为最大组合时，循环终止

ind = r-1 ] 找到最后一个元素

while lst[ind] == alpha[n-r+ind]: ] 使得  $lst[ind] \neq alpha[n-r+ind]$

ind -= 1

的  $r-1$  个元素

next = alpha.index(lst[ind]) + 1 ] 获得该元素在字母表中

for k, i in enumerate(range(ind, r)): ] 更新从  $lst[ind]$  到  $lst[r-1]$

更新  $lst[i] = alpha[next + k]$  的元素

yield lst

整个生成函数是生成所有可能的  $r$  组合

# Discrete Mathematics - P163

康托尔数字

对于正整数  $n$  和小于  $n!$  的非负整数  $0 \leq c < n!$

(有唯一的康托尔展开式  $c = a_1 \cdot 1! + a_2 \cdot 2! + \dots + a_{n-1} \cdot (n-1)!$ )

其中  $a_i \in [0, i]$ ,  $i = 1, 2, \dots, n-1$ .

则称  $\underline{\text{向量}}(a_1, a_2, \dots, a_{n-1})$  为  $c$  的康托尔展开

则对于  $\{1, 2, \dots, n\}$  的一个排列, 定义向量  $(b_1, b_2, \dots, b_{n-1})$

使得  $b_i$  为排列中小于  $i+1$  并且排在  $i+1$  之后的元素个数

可知对于  $b_i$ , 满足  $b_i \in [0, i]$ ,  $i = 1, 2, \dots, n-1$

于是有这个对应为从  $\{1, 2, \dots, n\}$  的排列到小于  $n!$  的非负整数的一个双射

证明过程有, 令集合  $R$  为  $\{1, 2, \dots, n\}$  的所有排列的集合

集合  $S$  为所有康托尔展开  $(a_1, a_2, \dots, a_{n-1})$  的集合

集合  $T$  为  $\{0, 1, 2, \dots, n! - 1\}$

则可知集合基数  $|R| = |S| = |T| = n!$

又对于  $R$  中的每一个不同的排列  $\underline{\text{合法的康托尔展开}}$

其生成的向量  $(b_1, b_2, \dots, b_{n-1})$  也不相同. 且  $(b_{n-1}, b_{n-2}, \dots, b_1)$  是

$(c_{n-1}, c_{n-2}, \dots, c_1)$  而对任意  $(b_1, b_2, \dots, b_{n-1})$ , 有  $(b_{n-1}, b_{n-2}, \dots, b_1) \in S$

于是存在一个映射  $f: R \rightarrow S$ , 且  $f$  是一一对应的.

$\& |R| = |S| = n!$ , 于是  $f$  是一一对应的

又存在  $g: S \rightarrow T$ , 且  $g$  是一一对应的

且即  $g \circ f$

于是可知从  $\{1, 2, \dots, n\}$  的排列到小于  $n!$  的非负整数存在一一对应

def cantor\_toTuple(lst): ] 将输入的排列转换成康托尔展开, 即  $f: R \rightarrow S$

ret, alpha = [], lst[:]

myQuickSort(alpha) 记录每个元素在字典中的大小排序

ind = list(map(lambda x: alpha.index(x), lst))

for i in range(0, len(lst)-1):

ret.append(sum(map(lambda x: x  $\in$  ind[i]  $\in$  ind[i+1])))] 元素的个数

return ret ret.reverse() / 即  $g: S \rightarrow T$  利用传入的字典和整数生成排列

def cantor\_toNum(lst): ] 将传入的康托尔展开

ret, n, temp = 0, len(lst), lst[:]

def cantor\_toList(alpha, n):

ret = [alpha[0]]

for ind, ele in enumerate(cantor\_expan(n)): ret.insert(ind, ele)

for ind, ele in enumerate(temp): ret = (n - ind) \* (ret + ele)

lencret - ele, alpha[ind+1])

= a<sub>1</sub> + 2! a<sub>2</sub> + ...

ret += alpha[lencret:]

+ (n-1)! a<sub>n-1</sub>

return ret

o = cantor\_toList

return ret