24w_assignment_01_report

November 19, 2024

Course	Combinatorial Algorithms
Semester	**2024 Winter**
Assignment	**01**
Group	**03**
Member 01	**999014681 Mingshan, LI**
Member 02	**999014772 Shunxi, XIAO**
Member 03	**999022064 Weizhi, LU**

0.1 Question 1. Permutations: Lexicographic Order

To implement in Python the functions `ranking`, `unranking` and `successor` for permutations in lexicographic, it is neccessary to implement the functions `dec2fac` and `fac2dec` at the first hand.

For an n-digit number in base factorial, say $f_{n-1} \dots f_2 f_1 f_{0_{(!)}}$, it is represented in the following code as a list-type vairable `fac_num` in python, such that for all `i` $\in \{0, 1, 2, \dots, n-1\}$, `fac_num[n-1-i]` $= f_i$.

[1]:
```
def factorial(n: int) -> int:
    if n == 0:
        return 1
    else:
        prod = 1
        i = 1
        while i <= n:
            prod = prod * i
            i = i + 1
        return prod

def dec2fac(dec_num: int, n: int) -> list:
    '''
    Transform a decimal number into an n-digit factorial base number.
    '''
    fac_num = [0] * n
    q, r, i = dec_num, 0, 1
    while q > 0:
        q, r = q // i, q % i
        fac_num[i-1] = r
```

```
        i = i + 1
    fac_num.reverse() # adapt to the way base-factorial number is represented
    return fac_num

def fac2dec(fac_num_: list, n: int) -> list:
    '''
    Transform an n-digit number in factorial base into a decimal number.
    '''
    fac_num = fac_num_[:] # we do not want to change arg. fac_num_
    fac_num.reverse()     # adapt to the way base-factorial number is␣
 ↪represented
    dec_num = 0
    prod = 1
    i = 1 # The first element in fac_num is always 0, so we begin with i = 1.
    while i < n:
        dec_num = dec_num + fac_num[i] * prod
        prod = (prod + 1) * prod
        i = i + 1
    return dec_num
```

Checking correctness:

[2]:
```
n = 4
n_fac = factorial(n)

for i in range(n_fac):
    fac_num = dec2fac(i, n)
    dec_num = fac2dec(fac_num, n)
    print(f'(i = {i:2})  In factorial: {fac_num} ,  In decimal: {dec_num:2}')
```

```
(i =  0)  In factorial: [0, 0, 0, 0] ,  In decimal:  0
(i =  1)  In factorial: [0, 0, 1, 0] ,  In decimal:  1
(i =  2)  In factorial: [0, 1, 0, 0] ,  In decimal:  2
(i =  3)  In factorial: [0, 1, 1, 0] ,  In decimal:  3
(i =  4)  In factorial: [0, 2, 0, 0] ,  In decimal:  4
(i =  5)  In factorial: [0, 2, 1, 0] ,  In decimal:  5
(i =  6)  In factorial: [1, 0, 0, 0] ,  In decimal:  6
(i =  7)  In factorial: [1, 0, 1, 0] ,  In decimal:  7
(i =  8)  In factorial: [1, 1, 0, 0] ,  In decimal:  8
(i =  9)  In factorial: [1, 1, 1, 0] ,  In decimal:  9
(i = 10)  In factorial: [1, 2, 0, 0] ,  In decimal: 10
(i = 11)  In factorial: [1, 2, 1, 0] ,  In decimal: 11
(i = 12)  In factorial: [2, 0, 0, 0] ,  In decimal: 12
(i = 13)  In factorial: [2, 0, 1, 0] ,  In decimal: 13
(i = 14)  In factorial: [2, 1, 0, 0] ,  In decimal: 14
(i = 15)  In factorial: [2, 1, 1, 0] ,  In decimal: 15
(i = 16)  In factorial: [2, 2, 0, 0] ,  In decimal: 16
(i = 17)  In factorial: [2, 2, 1, 0] ,  In decimal: 17
```

```
(i = 18)  In factorial: [3, 0, 0, 0] ,  In decimal: 18
(i = 19)  In factorial: [3, 0, 1, 0] ,  In decimal: 19
(i = 20)  In factorial: [3, 1, 0, 0] ,  In decimal: 20
(i = 21)  In factorial: [3, 1, 1, 0] ,  In decimal: 21
(i = 22)  In factorial: [3, 2, 0, 0] ,  In decimal: 22
(i = 23)  In factorial: [3, 2, 1, 0] ,  In decimal: 23
```

To implement the `ranking` and `unranking` algorithm, it is necessary to implement algorithms to transform from Lehmer codes to permutations and vice versa, respectively.

[3]:
```python
def permu2lehmer(permu: list) -> list:
    '''
    Transform a permutaiton into its lehmer code.
    '''
    n = len(permu)
    lehmer = [0] * n
    for i in range(n):
        curr = 0
        for j in range(i, n):
            if permu[j] < permu[i]:
                curr += 1
        lehmer[i] = curr
    return lehmer

def lehmer2permu(lehmer: list) -> list:
    '''
    Transform a lehmer code into the corresponding permtation.
    '''
    n = len(lehmer)
    num_list = [i for i in range(n)] # [0, 1, 2, ... , n-2, n-1]
    permu = [0] * n
    for i in range(n):
        curr = lehmer[i]
        permu[i] = num_list[curr]
        # eliminate the curr-th element from the num_list.
        if curr == 0:
            num_list = num_list[curr+1:]
        elif curr == n-1:
            num_list = num_list[:curr]
        else:
            num_list = num_list[:curr] + num_list[curr+1:]
    return permu
```

Checking correctness:

[4]:
```python
# Case 01
permu_01 = [2, 0, 3, 1]
lehmer_01 = permu2lehmer(permu_01) # it should be [2, 0, 1, 0]
```

```python
permu_01_ = lehmer2permu(lehmer_01)
print(f'permu: {permu_01} -> lehmer code: {lehmer_01} -> permu: {permu_01_}')

# Case 02
permu_02 = [3, 0, 1, 2]
lehmer_02 = permu2lehmer(permu_02) # it should be [3, 0, 0, 0]
permu_02_ = lehmer2permu(lehmer_02)
print(f'permu: {permu_02} -> lehmer code: {lehmer_02} -> permu: {permu_02_}')
```

```
permu: [2, 0, 3, 1] -> lehmer code: [2, 0, 1, 0] -> permu: [2, 0, 3, 1]
permu: [3, 0, 1, 2] -> lehmer code: [3, 0, 0, 0] -> permu: [3, 0, 1, 2]
```

Now, the `ranking` and `unranking` algorithms can be implemented as followings:

[5]:
```python
def permu_lex_rank(permu: list, n: int) -> int:
    '''
    Ranking algorithm of permutation over n in lexicographic order.
    '''
    r_fac = permu2lehmer(permu)  # rank in base factorial
    r_dec = fac2dec(r_fac, n)    # rank in base decimal
    return r_dec

def permu_lex_unrank(r_dec: int, n: int) -> list:
    '''
    Unranking algorithm of permutation over n in lexicographic order.
    '''
    r_fac = dec2fac(r_dec, n)   # rank in base factorial
    permu = lehmer2permu(r_fac) # permutation
    return permu
```

Checking correctness:

[6]:
```python
n = 4
n_fac = factorial(n)
for i in range(n_fac):
    curr_permu = permu_lex_unrank(i, n)
    curr_rank = permu_lex_rank(curr_permu, n)
    print(f'rank( {curr_permu} ) = {curr_rank:2}')
```

```
rank( [0, 1, 2, 3] ) =  0
rank( [0, 1, 3, 2] ) =  1
rank( [0, 2, 1, 3] ) =  2
rank( [0, 2, 3, 1] ) =  3
rank( [0, 3, 1, 2] ) =  4
rank( [0, 3, 2, 1] ) =  5
rank( [1, 0, 2, 3] ) =  6
rank( [1, 0, 3, 2] ) =  7
rank( [1, 2, 0, 3] ) =  8
rank( [1, 2, 3, 0] ) =  9
```

```
rank( [1, 3, 0, 2] ) = 10
rank( [1, 3, 2, 0] ) = 11
rank( [2, 0, 1, 3] ) = 12
rank( [2, 0, 3, 1] ) = 13
rank( [2, 1, 0, 3] ) = 14
rank( [2, 1, 3, 0] ) = 15
rank( [2, 3, 0, 1] ) = 16
rank( [2, 3, 1, 0] ) = 17
rank( [3, 0, 1, 2] ) = 18
rank( [3, 0, 2, 1] ) = 19
rank( [3, 1, 0, 2] ) = 20
rank( [3, 1, 2, 0] ) = 21
rank( [3, 2, 0, 1] ) = 22
rank( [3, 2, 1, 0] ) = 23
```

For the successor algorithm for permutation, we follow the following steps to implement the successor algorithm:

Given that $\pi : \{0, 1, \cdots, n-1\} \to \{0, 1, \cdots, n-1\}$ a permutation over $\{0, 1, \ldots, n-2, n-1\}$,

1. Find the largest $i \in \{0, 1, \ldots, n-1\}$ such that $\pi[i] < \pi[i+1]$. i D.N.E. $\iff$ π is the last permutation.
2. Find the largest $j \in \{0, 1, \ldots, n-1\}$ such that $\pi[k] < \pi[i] < \pi[j]$, $\forall k \in \{j+1, \ldots, n-1\}$ (i.e. j is the position of the last element among $\pi[i+1]$, $\ldots$, $\pi[n-1]$ that is greater than $\pi[i]$).
3. Interchange $\pi[i]$ and $\pi[j]$.
4. Reverse the sublist $[\pi[i+1], \pi[i+2], \ldots, \pi[n]]$.

[7]:
```python
def permu_lex_successor(permu_: list):
    permu = permu_[:] # we do not want to change arg. permu_
    n = len(permu)
    '''
    Step 1: Find the largest i such that  [i] <  [i+1].
    '''
    i = -1
    k = 0
    while k < n - 1:
        if permu[k] < permu[k+1]:
            i = k
        k = k + 1
    if i == -1: # such i D.N.E., that is, permu is the last permutation
        return [x for x in range(n)]
    '''
    Step 2: Find the largest j such that  [j] >  [i].
    '''
    j = i + 1
    l = i + 1
    while l < n:
```

```python
        if permu[l] > permu[i]:
            j = l
        l = l + 1
    '''
    Step 3: Swap  [i] and  [j]
    '''
    temp = permu[i]
    permu[i] = permu[j]
    permu[j] = temp
    '''
    Step 4:
    '''
    sublist = permu[i+1:]
    sublist.reverse()
    succ_permu = permu[:i+1] + sublist
    return succ_permu
```

Now, we check the correctness of `ranking`, `unranking` and `successor` algorithms.

[8]:
```python
N = 4
N_fac = factorial(N)
for r in range(N_fac):
    permu = permu_lex_unrank(r, N)
    rank = permu_lex_rank(permu, N)
    succp = permu_lex_successor(permu)
    if succp == permu_lex_unrank( (permu_lex_rank(permu, N) + 1) %  N_fac, N ):
        flag = True
    else:
        flag = False
    print(f'rank( {permu} ) = {rank:3},  successor( {permu} ) = {succp},␣
 ↪{flag}')
```

```
rank( [0, 1, 2, 3] ) =   0,  successor( [0, 1, 2, 3] ) = [0, 1, 3, 2], True
rank( [0, 1, 3, 2] ) =   1,  successor( [0, 1, 3, 2] ) = [0, 2, 1, 3], True
rank( [0, 2, 1, 3] ) =   2,  successor( [0, 2, 1, 3] ) = [0, 2, 3, 1], True
rank( [0, 2, 3, 1] ) =   3,  successor( [0, 2, 3, 1] ) = [0, 3, 1, 2], True
rank( [0, 3, 1, 2] ) =   4,  successor( [0, 3, 1, 2] ) = [0, 3, 2, 1], True
rank( [0, 3, 2, 1] ) =   5,  successor( [0, 3, 2, 1] ) = [1, 0, 2, 3], True
rank( [1, 0, 2, 3] ) =   6,  successor( [1, 0, 2, 3] ) = [1, 0, 3, 2], True
rank( [1, 0, 3, 2] ) =   7,  successor( [1, 0, 3, 2] ) = [1, 2, 0, 3], True
rank( [1, 2, 0, 3] ) =   8,  successor( [1, 2, 0, 3] ) = [1, 2, 3, 0], True
rank( [1, 2, 3, 0] ) =   9,  successor( [1, 2, 3, 0] ) = [1, 3, 0, 2], True
rank( [1, 3, 0, 2] ) =  10,  successor( [1, 3, 0, 2] ) = [1, 3, 2, 0], True
rank( [1, 3, 2, 0] ) =  11,  successor( [1, 3, 2, 0] ) = [2, 0, 1, 3], True
rank( [2, 0, 1, 3] ) =  12,  successor( [2, 0, 1, 3] ) = [2, 0, 3, 1], True
rank( [2, 0, 3, 1] ) =  13,  successor( [2, 0, 3, 1] ) = [2, 1, 0, 3], True
rank( [2, 1, 0, 3] ) =  14,  successor( [2, 1, 0, 3] ) = [2, 1, 3, 0], True
rank( [2, 1, 3, 0] ) =  15,  successor( [2, 1, 3, 0] ) = [2, 3, 0, 1], True
```

```
rank( [2, 3, 0, 1] ) =  16,  successor( [2, 3, 0, 1] ) = [2, 3, 1, 0], True
rank( [2, 3, 1, 0] ) =  17,  successor( [2, 3, 1, 0] ) = [3, 0, 1, 2], True
rank( [3, 0, 1, 2] ) =  18,  successor( [3, 0, 1, 2] ) = [3, 0, 2, 1], True
rank( [3, 0, 2, 1] ) =  19,  successor( [3, 0, 2, 1] ) = [3, 1, 0, 2], True
rank( [3, 1, 0, 2] ) =  20,  successor( [3, 1, 0, 2] ) = [3, 1, 2, 0], True
rank( [3, 1, 2, 0] ) =  21,  successor( [3, 1, 2, 0] ) = [3, 2, 0, 1], True
rank( [3, 2, 0, 1] ) =  22,  successor( [3, 2, 0, 1] ) = [3, 2, 1, 0], True
rank( [3, 2, 1, 0] ) =  23,  successor( [3, 2, 1, 0] ) = [0, 1, 2, 3], True
```

0.2 Question 2. Permutations: Minimal Change Order

It can be easily notices that, for any two different permutations π, π' over $[n] := \{0, 1, 2, \dots, n-1\}$, there are at least two positions $i_1, i_2 \in [n]$ such that $\pi[i_1] \neq \pi'[i_1]$ and $\pi[i_2] \neq \pi'[i_2]$.

The difference is minimal if the transposition is between two adjacent positions, i.e. $i_1 = i_2 + 1$ or vice versa.

Equivalently, we can say that, there exist an $i_0 \in \{1, 2, \dots, n-2, n-1\}$ such that $\forall j \in [n]$,

$$\pi'[j] = \begin{cases} \pi[i_0 + 1]\ , & j = i_0 \\ \pi[i_0]\ , & j = i_0 + 1 \\ \pi[j]\ , & j \neq i_0, i_0 + 1 \end{cases}$$

Next, we follow the steps below to implement the **Trotter-Johnson algorithm** for generating the list T^n of the permutations in $\prod^n$ in the minimal change order.

The **Trotter-Johnson algorithm** is implemented recursively:

1. **Base Case** : $T^1 = [[1]]$
2. **Inductive Step** : Suppose given T^{n-1} the list of the permutations in $\prod^{n-1}$ in the minimal order.
   - Create a new list T with the i-th n elements are n copies of T^{n-1}_i.
   - Every $t \in T$ is a k-th copy of some T^{n-1}_j, $j \in [n-1]$, $k \in [n]$. Let $b = j \mod 2$. Then, we replace $t \in T$ by the list obtained from inserting n in the $|(1-b)*(n+1)-(k-b)|$-th position of t.

[9]:
```python
def permu_trotterJohnson(n: int) -> list:

    '''
    Return all the permutations of {1, 2, ..., n}.
    '''

    curr_T = []
    if n == 1:
        curr_T = [[1]]
    else:
        prev_T = permu_trotterJohnson(n-1)
        for l in prev_T:
```

```
            i = 0
            while i < n:
                curr_T.append(l)
                i = i + 1
        n_fac = factorial(n)
        for i in range(n_fac):
            k = i % n + 1
            b = ( i // n ) % 2
            t = abs( (1-b) * n - (k-b) )
            curr_T[i] = curr_T[i][:t] + [n] + curr_T[i][t:]
    return curr_T
```

Checking correctness:

[10]:
```
N = 3
N_fac = factorial(N)
T_n = permu_trotterJohnson(N)
for i in range(N_fac):
    print(T_n[i])
```

```
[1, 2, 3]
[1, 3, 2]
[3, 1, 2]
[3, 2, 1]
[2, 3, 1]
[2, 1, 3]
```

Now, we turn our attention to the implementation of `ranking` algorithm for T^n.

Notice that, suppoes given $\pi \in T^n$ such that $\pi[k] = n$, then it must be the case that

$$\pi[:(k-1)] + \pi[(k+1):] := \pi' \in T^{n-1}$$

That is to say, $\pi \in T^n$ can be obtained from inserting n into the k-th position of some $\pi' \in T^{n-1}$. This means we can obtain $\text{rank}(\pi)$ recursively from the value $\text{rank}(\pi')$ and the specific $k \in [n]$, i.e. the position in π such that $\pi[k] = n$.

Moreover, we have the following recursive formula: $\forall n \in \mathbb{N}$,

$$\text{rank}(\pi, n) = \begin{cases} 0 & , \ n = 1 \ (\ i.e.\ \pi = [1]\ ) \\ n \cdot \text{rank}(\pi', n-1) + \epsilon & , \ n \geqslant 2 \end{cases}$$

where

$$\epsilon = \begin{cases} n - k & , \ \text{rank}(\pi', n-1) \text{ is even} \\ k - 1 & , \ \text{rank}(\pi', n-1) \text{ is odd} \end{cases}$$

[11]:

```
def epsilon(r: int, n: int, k: int) -> int:

    '''
    Implementation of the calculation of epsilon.

    Arguments:

    r: rank( ', n-1)
    n: the size of the permutation permu_
    k: the position in permu_ such that permu[k] = n
    '''

    if r % 2 == 0:
        return n - k
    else:
        return k - 1


def permu_colexi_rank(permu: list, n: int) -> int:

    '''
    The implementation of ranking alogorithmn of permutation over [n] = {1, 2, .
 ↪.., n}
    '''

    if n == 0:

        return 0

    else :

        k = permu.index(n)
        prev_r = permu_colexi_rank( permu[:k] + permu[k+1:], n - 1 )
        e = epsilon(prev_r, n, k+1) # k is obtained from python, which begins␣
 ↪from 0.
        return n * prev_r + e
```

Checking correctness:

[12]:

```
N = 3
permutations_N = permu_trotterJohnson(N)
for permu in permutations_N:
    r = permu_colexi_rank(permu, N)
    print(f'rank( {permu} ) = {r}')
```

```
rank( [1, 2, 3] ) = 0
rank( [1, 3, 2] ) = 1
rank( [3, 1, 2] ) = 2
```

```
rank( [3, 2, 1] ) = 3
rank( [2, 3, 1] ) = 4
rank( [2, 1, 3] ) = 5
```

Similarly, with the same idea, we implement `unranking` algorithm recursively.

It is an inverse procedure of `ranking` algorithm. So we formulate the following recursive formulas to generate the permutation of $[n]$ with rank r.

$$\text{unrank}(r, n) = \begin{cases} [1] & , \ n = 1 \\ \text{unrank}(r',\ n-1)[:k] + [n] + \ \text{unrank}(r',\ n-1)[k:] & , \ n \geqslant 2 \end{cases}$$

where

$$r' = \left\lfloor \frac{r}{n} \right\rfloor \text{ is the rank of } \pi' \text{ obtained from eliminateing } n \text{ from } \pi,$$

$$\epsilon = r - n \cdot r'$$

and

$$k = \begin{cases} n - \epsilon \ , & r' \text{ is even} \\ \epsilon + 1 \ , & r' \text{ is odd} \end{cases}$$

[13]:
```
def pos_k(r: int, e: int, n: int) -> int:

    '''
    Implementation of calculation of the position k, such that  [k] = n

    Arguments:
    r: the rank of  '

    '''

    if r % 2 == 0:
        return n - e
    else:
        return e + 1

def permu_colexi_unrank(r: int, n: int) -> int:

    '''
    The implementation of unranking alogorithmn of permutation over [n] = {1,␣
 ↪2, ..., n}
    '''

    if n == 1:

        return [1]
```

```
    else:

        prev_r = r // n
        e = r - n * prev_r
        k = pos_k(prev_r, e, n) - 1 # k is position in python list, which␣
↪begins from 0.
        prev_permu = permu_colexi_unrank(prev_r, n-1)

        return prev_permu[:k] + [n] + prev_permu[k:]
```

Checking correctness of `ranking` and `unranking` algorithm:

[14]:
```
N = 3
N_fac = factorial(N)
for r in range(N_fac):
    curr_permu = permu_colexi_unrank(r, N)
    curr_rank = permu_colexi_rank(curr_permu, N)
    print(f'rank( {curr_permu} ) = {curr_rank:2}')
```

```
rank( [1, 2, 3] ) =  0
rank( [1, 3, 2] ) =  1
rank( [3, 1, 2] ) =  2
rank( [3, 2, 1] ) =  3
rank( [2, 3, 1] ) =  4
rank( [2, 1, 3] ) =  5
```

Finally, the implementation of `successor` algorithm:

[15]:
```
def Parity(n, P):
    """
    Calculate the parity of a permutation.
    """
    a = [0] * n  # Array to track visited elements.
    c = 0  # Counter for the number of cycles.

    # Iterate through each element in the permutation.
    for j in range(n):
        if a[j] == 0:  # If this element is not visited yet.
            c += 1  # Increment the cycle count.
            a[j] = 1  # Mark it as visited.
            i = j

            # Traverse the cycle starting from this element.
            while(P[i] != j + 1):  # Continue until the cycle closes.
                i = P[i] - 1  # Move to the next element in the cycle.
                a[i] = 1  # Mark the element as visited.

    # Return the parity of the permutation.
```

```python
    return (n - c) % 2

def permu_colexi_successor(n, P):
    """
    Compute the next permutation in using a modified version of the Trotter-Johnson algorithm.
    """
    st = 0  # Starting index for the permutation.
    Q = P[:]  # Copy of the permutation to be used in calculations.
    done = False  # Flag to indicate if the next permutation is found.
    m = n  # Length of the current sub-permutation being considered.

    while((m > 1) and (not done)):
        d = 0  # Find the position of the largest element in the current range.
        while(Q[d] != m):  # Locate the position of 'm' in Q.
            d += 1

        # Temporarily remove m from Q by shifting elements to the left.
        for i in range(d, m - 1):
            Q[i] = Q[i + 1]

        # Check the parity of the resulting permutation.
        par = Parity(m - 1, Q)

        # Determine whether to swap or reduce m
        if(par == 1):  # Odd parity case.
            if d == m - 1:  # If 'm' is at the last position, reduce the size of the range.
                m -= 1
            else:  # Otherwise, swap 'm' with the next element to its right.
                temp = P[st + d]
                P[st + d] = P[st + d + 1]
                P[st + d + 1] = temp
                done = True  # Mark the next permutation as found.
        else:  # Even parity case.
            if(d == 0):  # If 'm' is at the first position, reduce the size of the range.
                m -= 1
                st += 1  # Update the starting index.
            else:  # Otherwise, swap 'm' with the previous element.
                temp = P[st + d]
                P[st + d] = P[st + d - 1]
                P[st + d - 1] = temp
                done = True  # Mark the next permutation as found.

        # Restore Q to its original state for the next iteration.
        Q = P[st:st + m]
```

```
    if m == 1 and not done:
        # return 'undefined'
        return [x+1 for x in range(n)]  # The input permutation is the last in␣
↪lexicographic order.
    else:
        return P  # Return the next permutation.
```

Checking the correctness of `ranking`, `unranking` and `successor` algorithms:

[16]:
```
N = 4
N_fac = factorial(N)
for r in range(N_fac):
    curr_permu = permu_colexi_unrank(r, N)
    curr_rank = permu_colexi_rank(curr_permu, N)
    next_permu = permu_colexi_successor(N, curr_permu[:])

    if next_permu == permu_colexi_unrank( (permu_colexi_rank(curr_permu, N) +␣
↪1) %  N_fac, N ):
        flag = True
    else:
        flag = False

    print(f'rank( {curr_permu} ) = {curr_rank:3},  successor( {curr_permu} ) =␣
↪{next_permu}, {flag}')
```

```
rank( [1, 2, 3, 4] ) =   0,  successor( [1, 2, 3, 4] ) = [1, 2, 4, 3], True
rank( [1, 2, 4, 3] ) =   1,  successor( [1, 2, 4, 3] ) = [1, 4, 2, 3], True
rank( [1, 4, 2, 3] ) =   2,  successor( [1, 4, 2, 3] ) = [4, 1, 2, 3], True
rank( [4, 1, 2, 3] ) =   3,  successor( [4, 1, 2, 3] ) = [4, 1, 3, 2], True
rank( [4, 1, 3, 2] ) =   4,  successor( [4, 1, 3, 2] ) = [1, 4, 3, 2], True
rank( [1, 4, 3, 2] ) =   5,  successor( [1, 4, 3, 2] ) = [1, 3, 4, 2], True
rank( [1, 3, 4, 2] ) =   6,  successor( [1, 3, 4, 2] ) = [1, 3, 2, 4], True
rank( [1, 3, 2, 4] ) =   7,  successor( [1, 3, 2, 4] ) = [3, 1, 2, 4], True
rank( [3, 1, 2, 4] ) =   8,  successor( [3, 1, 2, 4] ) = [3, 1, 4, 2], True
rank( [3, 1, 4, 2] ) =   9,  successor( [3, 1, 4, 2] ) = [3, 4, 1, 2], True
rank( [3, 4, 1, 2] ) =  10,  successor( [3, 4, 1, 2] ) = [4, 3, 1, 2], True
rank( [4, 3, 1, 2] ) =  11,  successor( [4, 3, 1, 2] ) = [4, 3, 2, 1], True
rank( [4, 3, 2, 1] ) =  12,  successor( [4, 3, 2, 1] ) = [3, 4, 2, 1], True
rank( [3, 4, 2, 1] ) =  13,  successor( [3, 4, 2, 1] ) = [3, 2, 4, 1], True
rank( [3, 2, 4, 1] ) =  14,  successor( [3, 2, 4, 1] ) = [3, 2, 1, 4], True
rank( [3, 2, 1, 4] ) =  15,  successor( [3, 2, 1, 4] ) = [2, 3, 1, 4], True
rank( [2, 3, 1, 4] ) =  16,  successor( [2, 3, 1, 4] ) = [2, 3, 4, 1], True
rank( [2, 3, 4, 1] ) =  17,  successor( [2, 3, 4, 1] ) = [2, 4, 3, 1], True
rank( [2, 4, 3, 1] ) =  18,  successor( [2, 4, 3, 1] ) = [4, 2, 3, 1], True
rank( [4, 2, 3, 1] ) =  19,  successor( [4, 2, 3, 1] ) = [4, 2, 1, 3], True
rank( [4, 2, 1, 3] ) =  20,  successor( [4, 2, 1, 3] ) = [2, 4, 1, 3], True
rank( [2, 4, 1, 3] ) =  21,  successor( [2, 4, 1, 3] ) = [2, 1, 4, 3], True
```

```
rank( [2, 1, 4, 3] ) =  22,  successor( [2, 1, 4, 3] ) = [2, 1, 3, 4], True
rank( [2, 1, 3, 4] ) =  23,  successor( [2, 1, 3, 4] ) = [1, 2, 3, 4], True
```