全排列以及相关算法

在程序设计过程中，我们往往要对一个序列进行全排列或者对每一个排列进行分析。全排列算法便是用于产生全排列或者逐个构造全排列的方法。当然，全排列算法不仅仅止于全排列，对于普通的排列，或者组合的问题，也可以解决。本文主要通过对全排列以及相关算法的介绍和讲解、分析，让读者更好地了解这一方面的知识，主要涉及到的语言是 C 和 C++。本文的节数：

由于本文的，内容比较多，所以希望读者根据自己的要求阅读，不要一次性读完，有些章节可以分开读。第 1 节到第 5 节提供了全排列的概念和一个初始的算法。第 6 节到第 8 节主要讲述了字典序的全排列算法。第 9 到第 10 节讲了有关字典序中中介数的概念。第 11 到第 12 节主要介绍了不同的中介数方法，仅供扩展用。第 13 节到 15 节介绍了邻位对换法的全排的有关知识。16 节讲了有关邻位对换法的中介数，仅供参考。第 17 节讲了组合数生成的算法。

1.全排列的定义和公式：

从 n 个数中选取 m（m<=n）个数按照一定的顺序进行排成一个列，叫作从 n 个元素中取 m 个元素的一个排列。由排列的定义，显然不同的顺序是一个不同的排列。从 n 个元素中取 m 个元素的所有排列的个数，称为排列数。从 n 个元素取出 n 个元素的一个排列，称为一个全排列。全排列的排列数公式为 n!，通过乘法原理可以得到。

2.时间复杂度：

n 个数（字符、对象）的全排列一共有 n!种，所以全排列算法至少时 **O(n!)**的。如果要对全排列进行输出，那么输出的时间要 **O(n*n!)**，因为每一个排列都有 n 个数据。所以实际上，全排列算法对大型的数据是无法处理的，而一般情况下也不会要求我们去遍历一个大型数据的全排列。

3.列出全排列的初始思想：

解决一个算法问题，我比较习惯于从基本的想法做起，我们先回顾一下我们自己是如何写一组数的全排列的：1，3，5，9（为了方便，下面我都用数进行全排列而不是字符）。

1，3，5，9.（第一个）
首先保持第一个不变，对 3，5，9 进行全排列。
同样地，我们先保持 3 不变，对 5，9 进行全排列。
保持 5 不变，对 9 对进行全排列，由于 9 只有一个，它的排列只有一种：9。接下来 5 不能以 5 打头了，5，9 相互交换，得到
1，3，9，5.
此时 5，9 的情况都写完了，不能以 3 打头了，得到
1，5，3，9
1，5，9，3
1，9，3，5
1，9，5，3
这样，我们就得到了 1 开头的所有排列，这是我们一般的排列数生成的过程。再接着是以 3、5、9 打头，得到全排列。**这里还要注意的一点是，对于我们人而言，我们脑子里相当于是储存了一张表示原有数组的表，1，3，5，9，1 开头的所有排列完成后，我们选择 3 开头，3 选完了之后，我们选择 5 开头，而不会再返过来选 1，而且知道选到 9 之后结束，但对于计算机而言，我们得到了 3，5，1，9 后，可能再次跳到 1 当中，因为原来数组的顺序它已经不知道了，这样便产生了错误。对于算法的设计，我们也可以维护这样一个数组，它保存了原始的数据，这是一种方法。同时我们还可以再每次交换后再交换回来，变回原来的数组，这样程序在遍历的时候便不会出错。**读者可以练习一下这个过程，思考一下你是如何进行全排列的，当然，你的方法可能和我的不太一样。
我们把上面全排列的方法归纳一下，基本上就是：任意选一个数（一般从小到大或者从左到右）打头，对后面的 n-1 个数进行全排列。聪明的读者应该已经发现，这是一个递归的方法，因为要得到 n-1 个数的全排列，我们又要先去得到 n-2 个数的全排列，而出口是只有 1 个数的全排列，因为它只有 1 种，为它的本身。写成比较规范的流程：
1.开始 for 循环。
2.改变第一个元素为原始数组的第一个元素（什么都没做）。
3.求第 2 个元素到第 n 个元素的全排列。
4.要求第 2 个元素到第 n 个元素的全排列，要递归的求第 3 个元素到第 n 个元素的全排列。
.....
5.直到递归到第 n 个元素到第 n 元素的全排列，递归出口。
6.将改变的数组变回。
7.改变第一个元素为原始数组的第二个元素。
（注：理论上来说第二次排列时才改变了第一个元素，即第 6 步应该此时才开始执行，但由于多执行一次无义的交换影响不大，而这样使得算法没有特殊情况，更容易读懂，如果一定要省时间可以把这步写在此处，这种算法我在下文中便不给出了，读者可以自己写。）

5.求第 2 个元素到第 n 个元素的全排列。
6.要求第 2 个元素到第 n 个元素的全排列，要递归的求第 3 个元素到第 n 个元素的全排列。
.....
5.直到递归到第 n 个元素到第 n 元素的全排列，递归出口。
6.将改变的数组变回。
.....
8.不断地改变第一个元素，直至 n 次使 for 循环中止。
为了实现上述过程，我们要先得到从第 m 个元素到第 n 个元素的排列的算法：

4.从第 m 个元素到第 n 个元素的全排列的算法：

```
void Permutation(int A[], int m, int n)
{
    if(m = = n)
        {
            Print(A);   //直接输出，因为前 n-1 个数已经确定，递归到只有 1 个数。
            return;
        }
    else
    {
        for(i=m;i<n;i++)     //进入 for 循环，对应第一步
        {
            swap(a[m],a[i]);   //交换，对应第二步
            Permutation(A,m+1,n);   //递归调用，对应三至五步
            swap(a[m],a[i]);   //交换，对应第六步
        }
}
```

为了使代码运行更快，Print 函数和 swap 函数直接写成表达式而不是函数（如果是 C++的话建议把 swap 写成内联函数，把 Print 写成宏）

```
void Permutation(int A[], int m, int n)
{
    int i, int temp;
    if(m = = n)
        {
            for(i = 0;i<n;i++)
            {
                if(i != n-1)
                    printf("%d ",A[i]); //有加空格
                else
                    printf("%d" A[i]);   //没加空格
            }  //直接输出，因为前 n-1 个数已经确定，递归到只有 1 个数。
            printf("\n");
            return;
        }
    else
    {
        for(i=m;i<n;i++)     /*进入 for 循环，对应第一步，注意此处是 m，而不是 0，因为
```

```
        是递归调用，对应的是第 m 个元素到第 n 个元素的全排列。*/
            {
                temp = A[m];
                A[m] = A[i];
                A[i] = temp; //交换，对应第二步
                Permutation(A,m+1,n);   //递归调用，对应三至五步
                temp = A[m];
                A[m] = A[i];
                A[i] = temp;
                  //交换，对应第六步
            }
        }
}
```

这个算法用于列出从第 m 个元素到第 n 个元素的所有排列，注意 n 不一定是指最后一个元素。算法的复杂度在于 for 循环和递归，最大的 for 是 n，递归为 n-1 所以为 O(n*n-1*n-2*...1) = O(n!).

对上述算法进行封装，便可以得到列出全排列的函数：

5.全排列算法：

```
void Full_Array(int A[],int n)
{
Permutation(A, 0,n);
}
```

如果读者仅仅需要一个全排列的递归算法，那么看到上面就可以了。下面将对全排列的知识进行扩充。

6.全排列的字典序：

字典序的英语一般叫做 dictionary order,浅显明白。

定义：**对于一个序列 a1,a2,a3,a4,a5....an 的两个排列 b1,b2,b3,b4,b5...bn 和 c1,c2,c3,c4,c5...cn, 如果它们的前 k（常数）项一样，且 c(k +1)> b(k+1),则称排列 c 位于排列 b（关于字典序）的后面**。如 1，2，3，4 的字典序排在 1，2，4，3 的前面（k=2），1，3，2，4 的字典序在 1，2，3，4(k=1)的后面。下面列出 1，2，3 按字典序的排列结果：

1，2，3
1，3，2
2，1，3
2，3，1
3，1，2
3，2，1

（有些读者会发现它们手写排列的时候也不自觉得遵照着这个规则以妨漏写，对于计算机也一样，如果有这样习惯的读者的话，那它们的实际算法更适合于表达为下面要讲的算法。）

定义字典序的好处在于，排列变得有序了，而我们前面的递归算法的排列是无序的，由同一个序列生成的不同数组（排列）如 1,2,3,4 和 2,3,4,1 的输出结果的顺序是不同的，这样便没有统一性，而字典序则可以解决这个问题。**很明显地，对于一个元素各不相同的元素集合，它的每个排列的字典序位置都不相同，有先有后**。

接下来讲讲如何求某一个排列的紧邻着的后一个字典序。对证明不感兴趣的读者只要读下面

加色的字即可。

定理：我们先来构造这样一个在它后面的字典序，再证明这是紧邻它的字典序。对于一个排列 a1,a2,a3...an,如果 a(n)> a(n-1)，那么 a1,a2,a3...a(n),a(n-1)是它后面的字典序，否则，也就是 a(n-1) > a(n),此时如果 a(n-2) < a(n-1)，那么在 a(n-1)和 a(n)中选择比 a(n-2)大的较小的那个数，和 a(n-2)交换，显然，它也是原排列后面的字典序。**更一般地，从 a(n)开始不断向前找，直到找到 a(m+1) > a(m)【如果 a(n)<a(n-1)，则找 a(n-1)和 a(n-2)，不断迭代，直到找到这样一组数或者 m=1 还不满足，则有 a(1) > a(2) > ...a(n)，是最大的字典序】，显然后面的序列满足 a(m+1)>a(m+2)>...a(n).找到 a(m+1)到 a(n)中比 a(m)大的最小的数，和 a(m)交换，并把交换后的 a(m+1)到 a(n)按照从小到大排序，前 m-1 项保持不变，得到的显然也是原排列后面的字典序，这个字典序便是紧挨着排列的后一个字典序。**

下面证明它是紧挨着的。1.如果还存在前 m-1 项和原排列相同并且也在原排列后面的字典序 a1,a2,a3...bm,...，bm>原 am，假设它在我们构造的字典序前面，那么必有 bm < 交换后的 am，但这是不可能的，因为 am 是后面序列中大于原来 am 的最小的一个，而 bm 必然又是后面序列中的大于 am 的一个元素，产生了矛盾。2.如果还存在前前 m 项和原排列相同并且也在原排列后面的字典序，它不可能在我们构造的字典序前面，因为我们对后面的数进行了升序排列，不存在比 a(m+1)还小的数。3.如果还存在前 k 项（k<m-1）和原排列相同并且也在原排列后面的字典序，它更不可能在我们构造的字典序前面，因为 b(k+1)>a(k+1)[k+1 < m]对于我们构造的字典序也满足。证明完毕。

证明完成后，我们便可以通过上述的构造方法求得一个排列的下一个字典序排列了。

7.求下一个字典序排列算法：

```
bool Next_Permutation(int A[], int n)
{
    int i,m,temp;
    for(i = n-2;i >= 0;i--)
        {
            if(A[i+1] > A[i])
            break;
        }
    if(i < 0)
        return false;
    m = i;
    i++;
    for(;i<n;i++)
        if(A[i] <= A[m])
            {
                i--;
                break;
            }
    swap(A[i],A[m]);
```

```
    sort(A+m+1,A+n);
    Print(A);
    return true;
}
```

swap 和 Print 函数读者可以自己写也可以参照我上面的写法，排序我这里直接使用了 C++标准库中的 sort，读者也可以自己写。有了这个算法后，我们便可以写一个非递归的列出全排列的方法，而且这个方法还带顺序：

```
void Full_Array(int A[],int n)
{
    sort(A,A+n);
    Print(A);
    while(Next_Permutation(A,n));
}
```

这个算法的时间复杂度为 **O(n^2 + n!*n) = O(n!*n)**[前面是排序的复杂度，后面是遍历 O（n!）乘以输出 n]。这个算法还有一个好处，那便是它可以处理元素相同的情况，不会重复输出。（之前的算法会重复，而且要注意的是，这个算法判断句中的> 、<有没有=号都是确定的，不能改，否则出现处理相同元素时便会陷入死循环，具体的写法读者可以自己举例判断，看看怎样会进入死循环。**如果要使之前递归的算法不重复，在交换之前要判断相邻着的两个数是否相同，如果相同，则不交换，比如和 A[i]交换，要判断 A[i]是否等于 A[i+1]**）。**不过这个算法的缺陷是它把原来数组给改变了**，读者可以自行在 Next_Permutation 当中使用 int* Array = new int(sizeof(A);或者 int* Array = malloc(sizeof(A)),然后把数组拷贝一遍，不对原数组进行处理，那么相应的全排列也要自己改写了，我这里就不写了。

8.C++ STL 库中的 next_permutation()函数：（#include<algorithm>）

幸运的是，C++已经给我们提供了 next_permutation 模版函数，所以不用自己写，也就不用担心死循环的问题，不过这个函数没有输出，而是直接把数组变成了它的下一个字典序。下面给出它的源代码：

```
// TEMPLATE FUNCTION next_permutation
template<class _BI> inline
    bool next_permutation(_BI _F, _BI _L)
    {
    _BI _I = _L;   //定义新的迭代器_I 并将尾地址赋给它。
    if (_F = = _L || _F = = --_I) //如果首地址等于尾地址或者等于尾地址小 1，直接返回 false
        return (false); //要注意是因为我们传递的是尾地址加 1（A+n = A[n]的地址），这个判断主要是考虑边界问题。
    for (; ; )   //死循环，用于找到 a(m+1) < a(m).
        {
        _BI _Ip = _I;   //定义迭代器_Ip 并将_I 赋给它。
        if (*--_I < *_Ip) //这里在比较 a(m+1)和 a(m)的大小，没找到则到下一个循环。如果
//找到，进入条件句，由于是用了--运算符，所以得到的实际上是_Ip，也即 a(m)。
            {
```

```
                    _BI _J = _L; //定义新的迭代器_J 并将尾地址赋给它，相当于从结尾开始
//找。前面我的算法是从 a(m+1)开始往后找，理论上从结尾开始找比较好，建议读者写的时
//候也从结尾往前找。
                for (; !(*_I < *--_J); ) //循环，直到找到 _I<_J，由于是减减，所以得到了第一
//个比_I 大的元素。由于是从结尾开始找，所以加了"！"，和我的相反。
                    ;//仅仅为了找而找。。。
                iter_swap(_I, _J);//a(m)和 a(i)交换，相当于我写的 swap 语句，注意传递的是迭
//代器，修改的是值。
                reverse(_Ip, _L);//相比于我的全排序，直接把 a(m+1)到 a(n)反序更有效率。
                return (true); //返回。
                }
            if (_I = = _F)      //判断是否到起点了，相应于 m+1 = 2，则把刚才反过来的反回去，
//再返回 false
                {
                    reverse(_F, _L);
//有些读者喜欢在反序之前判断是否到起点，而实际上到起点的情况只有一种（最后一个），
//不断地判断很浪费时间，还不如在最后再反回来。
                    return (false);
                }
            }
        }
```

它的第一个参数是迭代器（指针、数组）的首地址，第二个参数是末地址，可以这样传递：next_permutation(A,A+n)来获得整个数组的下个字典序。我在上面已经写了完整的解释，大家可以对比我的算法和 C++标准库里的算法，当然大家可以明显看到标准库算法的优越性，大家可以照着上面的解释自己写一个，不用全一样，模式一样即可，它的算法是最高效的（要不怎么能当模板？）。当然，标准库的算法为了使效率最快，安全性最高，总是喜欢用一些++啊--啊之类的运算符使得代码难读，读者在这方面可以不用模仿，算法上模仿就成了。下面再补充一些：在库中还有一个可以增加比较器的模版函数，不过实际上这个函数也不善于处理大型数据，有兴趣可以用。在 C++标准库中还提供了找上一个字典序的算法，perv_permutation()，用法一样，下面我附上原码，就不解释了，原理差不多，有兴趣的读者可以自己读读或者自己写一个。

```
// TEMPLATE FUNCTION prev_permutation
template<class _BI> inline
    bool prev_permutation(_BI _F, _BI _L)
    {_BI _I = _L;
    if (_F == _L || _F == --_I)
        return (false);
    for (; ; )
        {_BI _Ip = _I;
        if (!(*--_I < *_Ip))
            {_BI _J = _L;
            for (; *_I < *--_J; )
                ;
            iter_swap(_I, _J);
```

```
            reverse(_Ip, _L);
            return (true); }
        if (_I == _F)
            {reverse(_F, _L);
            return (false); }}}
```

9.字典序的中介数，由中介数求序号：

这里我要引入一个新的概念：**中介数**。为什么要引入这么一个概念呢？**很多时候，我们要通过一个排列得出它的字典序中的位置**（序号），比如 1234567 应该排在第 0 位（开始位），1234576 应该排在第 1 位，7654321 排在第 7！-1 = 5039 位。当然，我们可以通过计算 Next_Permutation 函数的迭代次数来得到这个数据，但这样时间复杂度最差为 O（n!），是非常不划算的，因为我们仅仅要求一个数的位置。所以我们要先从数学的角度去计算这个问题。

3456721 排在第几位？

1.先看看首位 3，在以 3 开头的数前面有以 2 开头和以 1 开头的数，这些数的个数为 2*6!个，其中 2 代表 1 和 2 两个数。

2.再看看第二位 4。如果已经确定由 3 开头，那么在 4 开头前面的数有几种呢？也是以 2 开头的数和以 1 开头的所有数，因为 3 已经放在开头，所以这里不算，这些数的个数为 2*5！。

3.再往下看，第三位 5.如果已经确定 34 开头，在 5 开头前面的有几种呢？由于 3、4 已经放在前面，这里还是只有 1、2.这些数的个数为 2*4！。

4.345 已经确定，再 6 开头的前面有 2*3！，再 7 开头前面有 2*2！，在 2 开头前面有 1*1！个（因为只剩 1 了）,在 1 开头前面的没有了。（0*0！）

于是我们得到在 3456721 前面的数有 2*6！+ 2*5！+ 2*4！+2*3！+2*2！+1*1 = 1745 位。也说是说，在 3456721 前面的数一共有 1745 位，由于我们是从 0 开始的，所以它的位置也就是 1745 位。

上述计算过程对应了相应的算法：从高位往低位看，或者对于数组而言从序号小的往大的看，按照 3，4，5，6，7，2，1[a[0],a[1],a[2],a[3],a[4],a[5],a[6]]的顺序，看它的右边比它小的数的个数 k[1],k[2],k[3]...（因为左边的数已经确定了），每一位的 **k[i]*[(n-i-1)]! [此处 n = 7]**乘以它所在的位数-1 的阶乘，比如 3 在第 7 位（在数组中是第 0 位，所以用 n-0），减 1 得到后面还剩 6 位，而 k[0]为 2，因为在 3 的右边比 3 小的数只有两个。其它的位依次类推。为了简化公式，**我们一般取 i=为数组序号加 1（即 a[0]我们认为是 a[1],用来和实际意义相近，因为我们实际中第一个数都是 1 而不是 C 语言中的 0）**。得到公式：

$$\sum_{i=1}^{n-1} k[i](n-i)!$$

其中 i 为数组的序号，从 1 开始，实际写算法时应从 0 开始，n 为数组元素的个数，k[i]表示第 i 个元素的右边（从第 i+1 个元素开始到数组末尾）比第 i 个元素小的元素的个数。

有了上述公式，就可以方便的计算一个排列在字典序中的位置了。比如 7654321

= 6 * 6！+ 5 * 5！ + 4 * 4! + 3 *3! + 2*2! + 1*1! = 5039.

到此，我们给出字典序中介数的定义：

定义：**由上述算法给出的数组 k[i]按照顺序排列所形成的数** $\overline{k[1]k[2]k[3]k[4]...k[n-1]}$ **叫做这个排列的字典序中介数**。如 7654321 的中介数为 654321,3456721 的中介数为 222221。中介数之所以称为中介数，是因为我们知道一个排列的中介数后，便可以很方便得求得它在字典序中的序号，而这个数在实际上没有明确的意义，所以称为中介数，要注意的是，**中介数**

比原来排列的位数（数组的元素个数）少一位，因为最后一位的结果必然是 0。读者应多多练习求一个排列的中介数以及相应的序号。以上算法的时间复杂度为$O(n^2)$，其中求中介数为$O(n^2)$，由中介数到序号为 O（n）。这个算法比初始的想法的 O(n!)要好的多。这个算法实际的代码我就不写了，读者可以自己写，这要是掌握这个算法的原理和过程。这里还要说的是这个算法不适合于有重复数字的情况，下面所讲的所有算法均不适合（可以想想为什么）。

10.由中介数求排列：

通过中介数可以很方便地求序号，而通过一个排列可以很方便地得到它的中介数，那么自然要想到另一个过程：通过中介数能否很方便地反推回排列呢？首先，显然地，中介数和排列是一一对应的，不存在一个中介数对应多个排列的情况（读者可以想想为什么）。所以，必然有办法可以求回去。按照我的习惯，还是从基本的想法做起，我们是怎样求得中介数的，由此反推回如何求排列。比如 222221。乍一看似乎很难入手，正如微分容易而积分难一样，当然，这个逆过程比积分要简单的多，因为我们已经确定它是唯一的。

方法：从最左边开始推算：2 开头，证明最高位的右边比它小的数只有两个，马上可以得到最高位是 3.第二位也是 2，因为 3 已经有了，那么还有比它小的两个数，那就是 4 了，以此类推，我们可以很方便地求得一个中介数的原排列。方法即从最左边推算。这是一个逆过程，最初看看是个不错的想法，做出来也比较快，而实际上不是这样的。我们看看我们实际的运算过程：最左边的数如果是 x1，那么它的原排列位是 x1+1，当我们去看第二个数时，我们先假想它是 x2+1，如果第一个排列位比 x2+1 来得大，那么没有问题，如果比它小或者相等，则要加上 1，所以实际上，我们每求一位数，都要先让它和前面的所有数进行比较来获得它的定位。求解中介数还有一些类似的方法，这里我便不罗列了，反正大同小异，表面上看起来比较简单，但实际实现起来是比较复杂的（由其是对计算机而言）。而且，在这种方法中，我们很难通过已知的排列序号反推出原来的排列，所以有必要给出新的中介数形式。

11.递增进位制数法：

为了改变由中介数求原排列的复杂性，我们要修改我们定义中介数的方法，我们的中介数是和相应的位数相关联的，中介数的下标对应了位的下标。我们定义一种新的中介数，称为递增进位制数，它的下标对应了数字大小（为了简便，下面还是叫做中介数，为了区分，我们把原来的中介数叫作初始中介数）。我们看一个新的数：3647521.它的初始中介数是 242321（读者应该已经会求了），我们不按照从左到右的顺序排，而是按照数字的大小来排，3 对应的是 2，因为是原排列位上的数是 3，所以把它放在第 3 位；6 对应的是 4，因为原排列位上的是 6，所以把它放在第 6 位，4 对应的是 2，放在第 4 位，7 对应 3，放在第 7 位，5 对应 2，放在第 5 位，2 对应 1，放在第 2 位，1 对应的是 0，我们没有写出来。我们得到 342221(0).相当于对初始中介数进行了排序，按照原来位上数字的大小排序，这样的数称为递增进位制数。读者可能还看不出来这个数和递增进位制这个名词有任何关系，它只是从中介数演变过来而已。实际上，经过这样的一个演变，每一位上的数都不会超过位的大小，比如第 4 位上的 2（我用蓝色记号的），它是不能达到 4 的，因为第 4 位对应原排列上的 4，而原排列上的 4 产生的初始中介数位是不可能大于 4 的（因为初始中介数是它右边比它小的数的个数，原排列是 4，不可能有 5 个比它小的数）。所以就形成了递增进位制数。它的第一位恒为 0，（所以用括号把 0 括号起来了），是个一进制数，第二位为 1 或 0，是个二进制数，第三位为 2 或 1 或 0，是个三进制数，以此类推。为了便于读者理解，我给出这样一个数的运算：342221 (0)+ 1 = 342221 + 1（因为一进制数加任何数都相当于把它进位）= 342300.因为第二

位是个二进制数，1+1 进位了，第二位为 0，第三位是个 3 进制数，2+1 还是进位了，第三位为 0，第四位是个四进制数，没有进位，所以为 3.得到 342300。**和字典序一样，对中介数加 1 后相当于对应的序号也加 1.这个过程也不是很难，原理还是加法，只是每一位的进制都不相同**，当然，如果做一个比较复杂的运算就比较难了。读者可以自己练习几个。由这样的**中介数求它的序号的方法**为

(((((k[1]*(n-1) + k[2])*(n-2)+k[3])*(n-3)..*2 + k[n-1].)，其中 k[i]就是新的中介数，i 从 1 开始表示从左往右数，即数组的序号（对应 n -i + 1 位）。这里的 n 表示的是原排列的元素个数。

实际上就是一种奇特的进制，**要注意的是，此时的序号并不是按照字典序排序的，而是一个新的排序方式**。1234576　新的中介数：100000。序号：720 而不是原来的 1。**但是范围是一样的，而且始末点是一样的**。1234567 ->0　7654321->5039(读者自己算一下)。讲了这么多，读者可能觉得这个方法看着非常麻烦，它到底哪里有利于求原排列呢？下面就讲它确定原排列的方法：**数空格法**。

＿＿＿＿＿＿＿我们画这样七个空格，来求 342221 的原排列：第 7 位数字是 3，我们从右向左数 3 个空格，再往前数一个空格，空格中填入对应的位数 7，把对应的空格擦除。

＿＿＿7＿＿＿。第 6 位数字是 4，我们从右向左数 4 个空格，其中已经放上数的不算空格，再往前数一个空格，空格中填入 6，把对应空格擦除。

＿6＿7＿＿＿。接下来的步骤也是一样的，第 5 位的数字是 2，数 2 个空格，再往前数一个空格填入 5。

＿6＿75＿＿。第 4 位是 2，数 2 个空格再往前数一个空格填入 4。

＿6475＿＿。第 3 位是 2，数 2 个空格再往前数一个空格填入 3.

36475＿＿。第 2 位是 1，数 1 个空格，再往前数一个空格填入 2.

364752＿。第 1 位是 0，数 0 个空格，再往前数一个空格填入 1.

3647521.由此，我们得到了原来的排列 3647521。可以看到，整个过程都是确定的，不需要进行额外的比较，相比于原来的中介数要简便得多，而且得到中介数的方法也基本类似，虽然它的本质是一个递增进制数可能有些困扰，如果不去管它的本质，则是以下简单的事情：

1.用原排列求新的中介序。

2.用中介序方便得求得原排列，使用空格法。

3.用中介序的一个公式求得新的序号。

接着还要讲讲递增中介数的最后一个好处，那便是可以方便得**通过序号求回中介数**。对于字典序来说，它是由几个阶乘相加得到的，反推回中介数基本上是不可能的（或者说是比较麻烦的），而对于递增进制中介数，这个过程非常的简单。通过上面的公式，我们很容易得到相逆的公式：记序号为 m.

m = m1*1 + k[n];　k[n]是一进制，恒为 0.

m1 = m2 * 2 + k[n-1]; 由进制的关系知道 k[n-1]是二进制的 0<=k[n-1]<=1

m2 = m3 * 3 + k[n-2]; 由进制的关系知道 k[n-1]是三进制的 0<=k[n-1]<=2

...

m(n-2) = m(n-1) * (n-1)+k[2]; 0<=k[2]<=n-2;

m(n-1) = k[1];

得到序号后，除以 2 取余得到 k[n-1],商接着除以 3 取余得 k[n-2]，以此类推得到原中介数。**由于递增进位制数法可以在原排列、中介数、序号之间进行较为方便的转换，我们可以从 0 开始递增中介数，进而转换为原排列并不断地求下一个序列，也可以得到全排列的算法。**

不过这里我就不写了，读者有兴趣可以写写。

12.递减进位制数法：

考虑到递增进制法中由于最低位是二进制，而且低位的进制都比较低，在求下一个排列时，中介数加 1 往往会导致很多的进位，计算比较麻烦，所以有了递减进位，**实际上就是将递增进位制的中介数倒过来即可**。如 342221 变成 122243.**求序号时的公式记住也是倒过来**。

((((k[1]*3) + k[2])*4+k[3])*5..k[n-2]*n + k[n-1]，其中 k[i]就是递减进制中介数，i 从 1 开始表示从左往右数，即数组的序号（对应 n -i + 1 位）。这里的 n 表示的是原排列的元素个数。

如 122243 表示为（（（（1*3+2）*4+2）*5+2）*6+4）*7+3 = 4735.

由于低位都是比较高的进制，所以不容易产生进位，求下一个排列非常地简单。

我们看这个例子：122243 + 1 = 122244.这里我们不需要再进位，而是很方便的计算。122244 对应的序号是 4736.反过来求出原排列的方法也很简单，先反序得到递增中介数再运算即可。

122244 = 442221(反序) = __ __ __ __ __ __ __ = __ __ 7 __ __ __ __ = __ 6 7__ __ __ __ = __ 6 7 __ 5__ __ = __ 6 7 4 5 __ __ = 3 6 7 4 5 2 1.（读者可以自行练习空格法）

和原来的 122243 对应的排列 3 6 4 7 5 2 1 相对比，发现仅仅是 4 和 7 交换了位置。我们可以再看看 122244 +1 = 122245 = 542221（反序）= __ __ __ __ __ __ __ = __ 7 __ __ __ __ __ = __ 7 6 __ __ __ __ = __ 7 6 __ 5 __ __ = __ 7 6 4 5 __ __ = 3 7 6 4 5 2 1. 我们可以发现它仅仅是上一个排列的 6 和 7 交换了位置。读者可能发现，如果再加两次，就会有进位，此时排列又更替了。所以，实际上，**递减进位制数法的排序采用的就是这种两个相邻元素交换的方式，并且是从右往左单向交换（至于它的数学原理这里就不深究了，有兴趣的读者可以研究一下）。通过递减进位制法也可以得到全排列的算法，相比于递增进位制法较简单**。

13.邻位对换法：

我们从递减进制数法从得到启示，递减进制法中的交换是单向的，这导致交换到头是排列会发生更替，如果我们采用双向的交换，便可以不断地交换下去，于是产生了**邻位对换法**。邻位对换法在找下一个排列的方法上在很多情况下要比字典序算法要快上许多，因为每次的下一个排列只是交换两个相邻的元素，**当然缺点是有到左端或者右端时要进行找最大可移动数的弊端，整体上速度和字面序算法差不多**。

当然，读者可能还没有理解邻位对换法的概念，下面开始讲解。在讲解之前先介绍数学中的两个概念，不过其实在这节里没有用，下面一节才用到。

定义 1：**在一个序列中任意两个元素对如果满足 a[k] < a[m](k < m)，则称为正序，否则称为逆序**。

定义 2：**如果一个排列中逆序的数目为奇数，则称为奇排列，若为偶数则称为偶排列**。

初始的时候，我们假设这个序列是一个升序的序列，而且最小元素至少为 1，升序的序列总是偶排列，并且我们设定初始所有数的移动（其实就是交换）的方向均从右向左。

我们给出可移动的概念：**如果一个数沿着它的方向的邻位比它小，则称这个数是可移动的**。由这个概念可以知道，1 是永远不可移动的，最大的数除非是在两端而且方向指向序列外面，要不一直是可移动的。

我们再规定一个性质：**如果一个可移动的数发生了移动，那么比它大的所有数的方向都反过来。对于一个排列而言，它的邻位对换法的下一个排列是最大的可移动的数移动后的结果**。

我们看一个例子：

1，2，3，4.[很显然，这是一个偶排列，因为它是升序序列。1<2,1<3,1<4,2<3,2<4,3<4,都是正序]。为了得到下一个排列，我们取最大的数 4，它是最大的可移动数，并把它向左移动：1，2，4，3.实际上是 3 和 4 的交换，这便是它的下一个排列。再移动：

1，4，2，3.再移动，
4，1，2，3.
由此我们得到了四个排列，每次都是通过交换相邻元素实现的。当 4 到头了之后，无法移动了，此时我们找到可移动的最大的数 3，并把它向左移动 1 次，得到
4，1，3，2.
此时由于 4 的移动方向已经反过来了，所以最大可移动数为 4，把 4 依次向右移动：
1，4，3，2
1，3，4，2
1，3，2，4.
4 到了头，再次无法移动了，此时最大的可移动的数变成了 3，把 3 向左移动一次，得到
3，1，2，4.
此时 4 的移动方向再反过来向左，得到
3，1，4，2.
3，4，1，2.
4，3，1，2.
此时 3 也到头了，此时我们找可移动的最大的数，2.得到
4，3，2，1.
4 和 3 的移动方向再反向右，
3，4，2，1
3，2，4，1
3，2，1，4.
4 到头后，由于此时 3 的移动方向向右，得到
2，3，1，4.
则 4 的方向又反，向左移动
2，3，4，1
2，4，3，1.
4，2，3，1.
此时 3 再向右移动，得到
4，2，1，3.
此时 4 再向右移动
2，4，1，3.
2，1，4，3.
2，1，3，4.
由此，我们从一个升序排列得到了全排列。通过这个例子，读者应该可以大概了解邻位对换法的基本过程了：
1.找到最大可移动数并移动至端点
2.找到现存的最大可移动数移动一次
3.回到原最大可移动数并移动至另一端点
4.找到现存的最大可移动数移动一次
……
5.找不到最大可移动数，循环结束，遍历结束。
由这个过程也可以直接得到一个非递归的算法，首先我们要写一个找到最大可移动数并移动一次的方法：

```
bool Movable(int A[],direct[],int n) //direct 参数用于接收每个元素移动方向的数组。
```

```
{
    int max = 1;//初始化最大可移动数为 1，因为规定 1 是最小的数，可以自己设定。
    int pos = -1;//初始化最大可移动数的位置为-1.
    /*下面先找到最大可移动数位置*/
    for(int i=0;i<n;i++)
    {
        if(a[i] < max)
            continue;
        if((i < n-1 && a[i] > a[i+1] && direct[i]) || (i > 0 && a[i] >a[i-1] && ! direct[i]))
            {
                max = a[i];
                Pos = i;
            }
    }
    /*下面对它进行移动*/
    if(pos = = -1)
        return false;
    if(p[pos])
        {
            swap(A[pos],A[pos+1]);
            swap(direct[pos],direct[pos+1]);//我这里用同样名字了，可以写模版，也可以改
        函//数名字

        }
    else
        {
            swap(A[pos],A[pos-1]);
            swap(direct[pos],direct[pos-1]);//我这里用同样名字了，可以写模版，也可以改
        函//数名字

        }
/*最后调整所有比最大可移动数大的数的方向*/
    for(int i = 0;i<n;i++)
    {
        if(A[i] > max)
            direct[i] = !direct[i];
    }
    return true;
}
```

有了这个方法后，便可以进行全排列了。

17.邻位对换法全排列：

```
void Full_Array(int A[], int n)
{
    bool* direct = new bool[n]; //产生一个记录每个元素移动方向的数组
```

```
    sort(A,A+n); //将原序列变成一个升序
    for(int i=0;i<n;i++)
        direct[i] = false; //初始化移动方向为 false，表示从右向左。

    do
    {
        Print(A,n);
        if(A[n-1] = = n)
            for(int i=n-1;i>0;i--)
            {
                swap(A[i],A[i-1]);
                swap(direct[i],direct[i-1]); //我这里用同样名字了，可以写模版，也可以改
            函//数名字
                Print(A,n);
            }
        else
            for(int i=0;i < n; i++)
            {
                swap(A[i],A[i+1]);
                swap(direct[i],direct[i+1]); //我这里用同样名字了，可以写模版，也可以改
            函//数名字
                Print(A,n);
            }

      }while(Movable(A,direct,n));
    delete []direct;
}
```

15.邻位对换法的下一个排列：

为了让读者更容易理解和掌握邻位对换法的基本过程，我先通过一个例子让大家了解如何去得到下一个排列，并接着给出了相应的算法，但是，如果我们得到了一个排列比如：2，3，4，1.**我们仅仅知道通过一次邻位交换可以得到它的下一个排列，但是是哪一位的交换呢？**

首先，我们得找到最大的数的位置。2，3，4，1.中最大的数位于第 3 个位置。位置是好找的，那方向如何判断呢？

我们接着要判断最大的数此时的移动方向。之前给出了奇排列和偶排列的概念，在这里便用的上了。**每当我们的最大的数从一端移到另一端时，就要进行最大可移动数的交换，这个过程过后，在不考虑最大数的数列中便会增加一个逆序**。而**初始的** 1，2，3 的**逆序为 0**，最大的数在移动的过程中不会改变除去最大数后的数列的顺序，**所以，不考虑最大数后的数列的逆序为偶数个时（偶排列），最大数在向左移动，逆序为奇数个时（奇排列），最大数向右移动**。2，3，4，1 中不考虑最大数的序列 2，3，1 的逆序为 1，所以 4 在向右移动。**同样地，如果最大数此时在某一端点，我们可以通过上述性质判断是移动最大数还是找到最大可移动数并进行交换**。当然这里我们还有问题没有解决：**如何判断非最大数的数此时的方向呢？**如果无法判断这个，还是无法找到下一个排列。

与上面类似地，我们可以得到更一般的结论：**一个非最大数的方向由所有比它小的数构成的序列的逆序决定的，如果是偶数个时，向左，奇数个时向右[也即的所有的数包括最大数都满足这条规则]**。

2，3，4，1 中容易知道此时 3 是向右的，2 是向左的，4 是向左的，1 是不动的。读者可以自己对 1，2，3，4 这个全排列的每一个进行练习。

16.邻位对换法的中介数：

邻位对换法的中介数也不难求。对于某个排列如 2341，**，从 1 开始看起（1 没有方向，不用算），如果它是向左的，则右边所有比它小的数的个数为中介数的最高位，如果是向右的，则左边所有比它小的数的个数为中介数的最高位，位逐次降低**，如 2 对应 k[1] = 1，最高位为 1，3 对应 k[2] = 1，第三位为 1，4 是向左的，第二位为 k[3] = 1.

得到 111 为 2341 的中介数。

对应的序号的求法为

(1 * 3 + 1) * 4 + 2 = 17，和递减进位的公式一样，所以通过序号求中介数也是和递减进位、递增进位类似，除以 n 取余，除以 n-1 除余，。。。。。。

邻位对换法由中介数求原排列的方法似乎也不是很简单，读者就不用掌握了，这里就概括一下：首先要判断最大数的方向，这个和上面一样，要由其它数构成的序列的奇偶性决定，所以要求其它数构成序列的奇偶性，这个奇偶性通过求序号的公式来得到，由**性质原排列中取所有小于等于 k 的数构成的数列（顺序不变）的奇偶性和对应的序号的奇偶性相同**。我举简单的例子：2341 的中介数为 112.那么 231 的中介数为 11（舍掉低相应位数即可得到相应的中介数），则它的序号为 1*3+**1** = 4 是偶数，所以最高位的方向 4 的方向向左。以此类推即可。

17.组合数的字典序与生成：

最后我们讲讲组合数的生成。首先我们看看列出集合的所有子集、要列出一个集合{1,2,3,4}的所有子集是很容易的，我们可以按照二进制数的顺序，0000，0001，0010，0011，0100，0101，0110，0111.....来表示我们要取的元素，其中 0 表示不取，1 表示取，这样就获得了一个顺序。而组合也包含在这个顺序当中。我们看从{1,2,3,4}中选取两个元素的所有组合：

0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111.

蓝色标记的是我们所取的组合。对应的顺序便是 {3，4}{2，4}{2，3}{1，4}{1，3}{1，2}我们可以**用字典序对这些组合进行排序**：{1,2}{1,3}{1,4}{2,3}{2,4}{3,4}.共 6 种。

按照我的习惯，我们先看看我们平常列举组合数所采取的策略。比如说 1，2，3，4，5 取 3 个元素的组合。我们先从小的取：1，2，3.再把最后一个数改变:1,2,4;1,2,5.当到达 5 之后，含有 1，2 的所有组合已经被罗列完了。改变 2 为 3，1，3，4；1，3，5；此时我们不能再选择 2 进行罗列，否则会重复。改变 3 为 4，此时 2 和 3 都不能放入，1，4，5.改变 4 为 5 时，2，3，4 都不能放入，不存在组合了。这时改变 1 为 2。之后不能再选择 1 罗列。所以，对于一个组合 C1C2C3C4...Cr(每个代表一个数，一共 r 个数，代表一个组合，数从[1,n]中选)，**由于一个组合自身是不讲顺序的，我们可以对组合进行升序排列，使得 C1<C2<C3<...Cr**。

一开始我们取最小的元素的组合，并按照字典序把 Cr 慢慢递增 1，这显然是在组合不断取字典序的下一个组合，没有夹杂在这这个方法能取到的值中间的其它组合。在所有组合数当 Cr 到达 n 之后，Cr 无法变大了，就要做其它的事。由于强制了升序，C(r-1) < Cr ，所以 C(r - 1) 到达 n-1 后也得去做其它的事。得到性质：

C(r - m)到达 n - m 后就不能再递增上去。由 k = r - (r - k)得到

C(k)到达 n - (r - k) = n - r + k 后就不能再递增上去。

那么，对于 C1C2C3C4...Cr，我们从 Cr 开始向前找，直到找到有 Ci < n - r + i，并执行 Ci = Ci + 1。由于所有的组合都已经强制升序，所以 Cj = Ci + (j-i),j = i+1,i+2,...r 即它后面

的每一项都比前一项大 1。由于 Ci < n -r + i，Cr = Ci + (r - i) <=n(因为 Ci 比原来大了 1)，所以构造出来的新的 C1C2C3C4...Cr 仍是[1,n]中选 r 个元素的一个组合，而且显然它在原组合的字典序后。 由此，我们构造出了一个排在原组合的字典序后面的组合。接下来证明它是紧邻着原组合的。

证明：假设有一个组合 B1B2B3B4...Br 的前 i-1 个元素和 C1C2C3C4...Cr 一样，且在原组合字典序后，在我们构造的字典序前，那么必有新 Ci > Bi > 原 Ci，这是不可能的，因为新 Ci = 原 Ci + 1。同样不可能存在前 i 个元素一样但在新 Ci 字典序前的组合，因为强制排序后 Cj 到 Cr 的每个元素都是最小的。同样不可能存在前 k 个元素一样（k < i）但在新 Ci 字典序前的组合。

如果找不到有 **Ci < n - r + i**，说明已经到达最后一个：n -r + 1,n - r + 2,...n.由此我们可以直接得到组合数的相对字典序，下一个组合的算法：

```
bool Next_Combination(int A[],int n,int r)
{
    int i;
    sort(A,A +r);
    for(i = r- 1;!(A[i] < n - r + i) && i > 0;i--);
    if (i = = 0)
        return false;
    A[i] = A[i] + 1;
    for(int j=i+1;j<r;j++)
        A[j] = A[i] + j - i;
    return true;
}
```

当然，这个算法中的组合是从 1 到 n 中选取的。读者可以通过给自己写选取的组合规定序号进行选取，也可以自行修改算法。

————————by 夜せ丨深