http://www.cnblogs.com/devymex/archive/2010/08/17/1801122.html

**概念**

全排列的生成算法有很多种，有递归遍例，也有循环移位法等等。C++/STL中定义的next\_permutation和prev\_permutation函数则是非常灵活且高效的一种方法，它被广泛的应用于为指定序列生成不同的排列。本文将详细的介绍prev\_permutation函数的内部算法。

按照STL文档的描述，next\_permutation函数将按字母表顺序生成给定序列的下一个较大的序列，直到整个序列为减序为止。prev\_permutation函数与之相反，是生成给定序列的上一个较小的序列。二者原理相同，仅遍例顺序相反，这里仅以next\_permutation为例介绍算法。

下文内容都基于一个假设，即序列中不存在相同元素。对序列大小的比较做出定义：两个长度相同的序列，从两者的第一个元素开始向后比较，直到出现一个不同元素（也可能就是第它们的第一个元素），该元素较大的序列为大，反之序列为小；若一直到最后一个元素都相同，那么两个序列相等。

设当前序列为pn，下一个较大的序列为pn+1，那么不存在pm，使得pn < pm < pn+1。

**问题**

给定任意非空序列，生成下一个较大或较小的序列。

**数学推导**

根据上述概念易知，对于一个任意序列，最小的序列是增序，最大的序列为减序。那么给定一个pn要如何才能生成pn+1呢？先来看下面的例子：

我们用<a1 a2 ... am>来表示m个数的一种序列。设序列pn=<3 6 4 2>，根据定义可算得下一个序列pn+1=<4 2 3 6>。观察pn可以发现，其子序列<6 4 2>已经为减序，那么这个子序列不可能通过交换元素位置得出更大的序列了，因此必须移动最高位3（即a1）的位置，且要在子序列<6 4 2>中找一个数来取代3的位置。子序列<6 4 2>中6和4都比3大，但6大于4。如果用6去替换3得到的序列一定会大于4替换3得到的序列，因此只能选4。将4和3的位置对调后形成排列<4 6 3 2>。对调后得到的子序列<6 3 2>仍保持减序，即这3个数能够生成的最大的一种序列。而4是第1次作为首位的，需要右边的子序列最小，因此4右边的子序列应为<2 3 6>，这样就得到了正确的一个序列pn+1=<4 2 3 6>。

下面归纳分析该过程。假设一个有m个元素的序列pn，其下一个较大序列为pn+1。

1) 若pn最右端的2个元素构成一个增序子序列，那么直接反转这2个元素使该子序列成为减序，即可得到pn+1。

2) 若pn最右端一共有连续的s个元素构成一个减序子序列，令i = m - s，则有pn(i) < pn(i+1)，其中pn(i)表示排列pn的第i个元素。例如pn=<1 2 5 4 3>，那么pn的右端最多有3个元素构成一个减序子集<5 4 3>，i=5-3=2，则有pn(i)=2 < 5=pn(i+1)。因此若将pn(i)和其右边的子集s {pn(i+1), pn(i+2), ..., pn(m)}中任意一个元素调换必能得到一个较大的序列（不一定是下一个）。要保证是下一个较大的序列，必须保持pn(i)左边的元素不动，并在子集s {pn(i+1), pn(i+2), ..., pn(m)}中找出所有比pn(i)大的元素中最小的一个pn(j)，即不存在pn(k) ∈ s且pn(i) < pn(k) < pn(j)，然后将二者调换位置。现在只要使新子集{pn(i+1), pn(i+2), ..., pn(i), ...,pn(m)}成为最小序列即得到pn+1。注意到新子集仍保持减序，那么此时直接将其反转即可得到pn+1 {pn(1), pn(2), ..., pn(j), pn(m), pn(m-1), ..., pn(i), ..., pn(i+2), pn(i+1)}。

**复杂度**

最好的情况为pn的最右边的2个元素构成一个最小的增序子集，交换次数为1，复杂度为O(1)，最差的情况为1个元素最小，而右面的所有元素构成减序子集，这样需要先将第1个元素换到最右，然后反转右面的所有元素。交换次数为1+(n-1)/2，复杂度为O(n)。因为各种排列等可能出现，所以平均复杂度即为O(n)。

**扩展**

1. 能否直接算出集合{1, 2, ..., m}的第n个排列？

设某个集合{a1, a2, ..., am}（a1<a2<...<am）构成的某种序列pn，基于以上分析易证得：若as<at，那么将as作为第1个元素的所有序列一定都小于at作为第1个元素的任意序列。同理可证得：第1个元素确定后，剩下的元素中若as'<at'，那么将as'作为第2个元素的所有序列一定都小于作为第2个元素的任意序列。例如4个数的集合{2, 3, 4, 6}构成的序列中，以3作为第1个元素的序列一定小于以4或6作为第1个元素的序列；3作为第1个元素的前题下，2作为第2个元素的序列一定小于以4或6作为第2个元素的序列。

推广可知，在确定前i（i<n）个元素后，在剩下的m-i=s个元素的集合{aq1, aq2, ..., aq3}（aq1<aq2<...<aqm）中，以aqj作为第i+1个元素的序列一定小于以aqj+1作为第i+1个元素的序列。由此可知：在确定前i个元素后，一共可生成s!种连续大小的序列。

根据以上分析，对于给定的n（必有n<=m!）可以从第1位开始向右逐位地确定每一位元素。在第1位不变的前题下，后面m-1位一共可以生成(m-1)!中连续大小的序列。若n>(m-1)!，则第1位不会是a1，n中可以容纳x个(m-1)!即代表第1位是ax。在确定第1位后，将第1位从原集合中删除，得到新的集合{aq1, aq2, ..., aq3}（aq1<aq2<...<aqm），然后令n1=n-x(m-1)!，求这m-1个数中生成的第n1个序列的第1位。

举例说明：如7个数的集合为{1, 2, 3, 4, 5, 6, 7}，要求出第n=1654个排列。

(1654 / 6!)取整得2，确定第1位为3，剩下的6个数{1, 2, 4, 5, 6, 7}，求第1654 % 6!=214个序列；

(214 / 5!)取整得1，确定第2位为2，剩下5个数{1, 4, 5, 6, 7}，求第214 % 5!=94个序列；

(94 / 4!)取整得3，确定第3位为6，剩下4个数{1, 4, 5, 7}，求第94 % 4!=22个序列；

(22 / 3!)取整得3，确定第4位为7，剩下3个数{1, 4, 5}，求第22 % 3!=4个序列；

(4 / 2!)得2，确定第5为5，剩下2个数{1, 4}；由于4 % 2!=0，故第6位和第7位为增序<1 4>；

因此所有排列为：3267514。

2. 给定一种排列，如何算出这是第几个排列呢？

和前一个问题的推导过程相反。例如3267514：

后6位的全排列为6!，3为{1, 2, 3 ,4 , 5, 6, 7}中第2个元素（从0开始计数），故2\*720=1440；

后5位的全排列为5!，2为{1, 2, 4, 5, 6, 7}中第1个元素，故1\*5!=120；

后4位的全排列为4!，6为{1, 4, 5, 6, 7}中第3个元素，故3\*4!=72；

后3位的全排列为3!，7为{1, 4, 5, 7}中第3个元素，故3\*3!=18；

后2位的全排列为2!，5为{1, 4, 5}中第2个元素，故2\*2!=4；

最后2位为增序，因此计数0，求和得：1440+120+72+18+4=1654

**C++/STL实现**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | #include <algorithm>  #include <iostream>  #include <string>  using namespace std;  //主函数，算法详见相关说明  int main(void) {      //循环处理输入的每一个字符串      for (string str; cin >> str;) {          if (str.empty()) {              continue;          }          //如果字符串只有1个字符，则直接输出结束          if (str.length() <= 1) {              cout << "No more Permutation" << endl;          }          //iPivot为右边最大减序子集左边相邻的一个元素          string::iterator iPivot = str.end(), iNewHead;          //查找右边最大的减序子集          for (--iPivot; iPivot != str.begin(); --iPivot) {              if (\*(iPivot - 1) <= \*iPivot ) {                  break;              }          }          //如果整个序列都为减序，则重排结束。          if (iPivot == str.begin()) {              cout << "No more Permutation" << endl;          }          //iPivot指向子集左边相邻的一个元素          iPivot--;          //iNewHead为仅比iPivot大的元素，在右侧减序子集中寻找          for (iNewHead = iPivot + 1; iNewHead != str.end(); ++iNewHead) {              if (\*iNewHead < \*iPivot) {                  break;              }          }          //交换iPivot和iNewHead的值，但不改变它们的指向          iter\_swap(iPivot, --iNewHead);          //反转右侧减序子集，使之成为最小的增序子集          reverse(iPivot + 1, str.end());          //本轮重排完成，输出结果          cout << str << endl;      }      return 0;  } |