

自然的组合算法

Natural Combination Algorithm

BenBear
benbearchen@gmail.com

2006 年 4 月

关键词：组合，排列，字典序，STL，枚举

Natural Combination Algorithm 是字典序的组合算法，可以线性地遍历从一些元素中选出若干元素的所有组合。此算法颇为简单，在实现上十分简约。大致思想是找出相邻两组组合之间的交叉点，然后经简单调整即可。可配合SGI C++ STL 中的 **permutation** 算法（本算法即从这个算法参考而来）。

1 一些表示方法

在正式说明算法以前，需要对组合的表示方式进行说明。本算法对参加组合的所有元素分为两个区间，被选入组合的元素一起，称为区间 P ；其它元素一起称为区间 Q 。 P 与 Q 的元素的合集即是全体元素的集合。此处所说的区间，可以看成集合，只是允许存在相同元素，而且是升序的。

两个区间在程序里，用升序的线性表来存放，在处理之时，也使用此顺序。在 C++ STL 中，“区间”习惯以左闭右开的 $[first, last)$ 表示， $first$ 指向区间（线性表）的第一个元素； $last$ 指向区间的最后一个元素的下一个元素，相当于追加新元素时的插入位置。把 $first$ 、 $last$ 这样的东西称为指示器（C++ 正式名称为 **iterator**）， $first$ 、 $last$ 称为 $[first, last)$ 的首指示器与尾指示器。用 $*iter$ 表示指示器 $iter$ 指向的实际元素，但注意区间的尾指示器指向的元素不属于区间（即 $*last$ 不属于区间 $[first, last)$ ）。

特别地，此算法将 Q 追接到 P 后面。所以 Q 的首指示器即等于 P 的尾指示器，因而算法中 P 与 Q 也写成 $[first, middle)$ 与 $[middle, last)$ ；这样在程序的参数传递时，以参数列表 $(first, middle, last)$ 即可表示。

组合的元素本来是无序的，但在此将组合内的元素按升序排序之后，再将这些组合之间按照字典序来排定先后顺序。例如从 $\{1, 2, 3, 4, 5\}$ 中选出三个元素，其所有组合的字典顺序如表1所示。

Table 1: {1, 2, 3, 4, 5} 中五选三

序号	选入元素 (P)	选出元素 (Q)
1	1 2 3	4 5
2	1 2 4	3 5
3	1 2 5	3 4
4	1 3 4	2 5
5	1 3 5	2 4
6	1 4 5	2 3
7	2 3 4	1 5
8	2 3 5	1 4
9	2 4 5	1 3
10	3 4 5	1 2

2 从英文字典说起

考察一本英文字典的单词的排列顺序，可以发现几个特征：一、前后单词之间的变化，尽可能从后面开始；二、变化之时，能变到b就不变到c。简言之就是从后从小。

/*用权值的观点来看，一个单词有位置的权值与字母的权值两种。
位置的权值：后一位置的权值远小于前字母权值，且以第一个位置为一个单位。字母的权值：a 到z 的权值递增，但其增长程度较位置权值的变化要小得多。所有“位置与其上字母的权值之积”的和，作为一个单词的权重。而英文字典其实就是按照单词的权重从小到大排序。如果把单词前加上“0.”，则可将其看成X 进制的小数来比较大小了。*/

同样作为字典序，我们可以用类似的方法来考察前面五选三十个组合的顺序。注意带下划线的位置即是上一组合以来，变化的起点。考虑{1, 2, 3} 中的2 可以变化到3，而3 可变化到4，但3 在后，所以选择将3 变化到4。考虑{1, 2, 3} 中的3 可以变化到4 或5，显然变化到4 比变化到5 要更小一点。考虑{1, 2, 5}，最右可变化的是2（可变化到3 或4），选择变到3；根据升序的原则，第三个位置的数只能是4 或5，当然选择更小的4。

思考字典序变化的规律，可用三步表示：

1. 找出最后面的可以增长的位置，称此位置为“切点”；
2. 将此位置做一个最小的增长；
3. 在后面接一个最小的尾巴。

这三个步骤都是可以证明的。但是由于表达能力有限，只有依赖诸位的直觉了。不过在后面有少许论断，但都不严格。

另外，可以参考下面这个简单的输出组合的程序，可更加明了其变化。

```
for a := 1 to 5-2 do
  for b := a+1 to 5-1 do
    for c := b+1 to 5 do
      WriteLn ('One combination is:', a, b, c);
```

/*在这个程序里，很容易看出，先从 c 增长一个单位，也就是 1；如果不可行，再从 b 增长 1；如果还不可行，那就从 a 增长 1；再不行的话，就表示排列结束。在某个变量增长之后，其后的变量在它的基础累加 1 而得到值。由 a、b、c 之间的递增性可知，第一步是找出最大的可增长的变量；然后增长 1，相当于找出最小的比他大的元素；然后再用尽可能小的值来填充剩下的变量。*/

3 枚举型的算法

现在需要对算法表述一下了。这个“自然”的字典序的组合算法，通过枚举当前的组合的字典序的下一个组合，来达到遍历所有组合的目的。比如{1,2,3} 枚举可以得到{1,2,4}，{1,2,4} 枚举可以得到{1,2,5}，其他类似。特殊的，另字典序的最后一个组合通过此算法得到字典序的的第一个组合，称之为新一轮的结束与新一轮的开始。

使用本算法遍历组合的一般步骤是：

```
初始化到字典序的第一个组合；
do
{
  处理当前组合；
  调用本算法求得下一个组合；
}
while (字典序未结束)；
```

4 算法分析

下面对一般性的组合的增长变化进行分析。特殊地，现在不考虑存在重复元素的情况，待得出结论之后，再推论至存在重复元素的情况。

判断 P 中的某个位置是否可以增长，需要与 Q 中的元素比较。这是因为如果有 $*a(\in P)$ 可以变到 $*b(\in P)$ 而保持组合的增长，那么必定有 $*c(\in Q) > *b$ 被选入 $*b$ 的尾巴 ($*b$ 前移到 $*a$ ，则后面必定出现空缺，只有从 Q 中补足)。既然如此，那么肯定是把 $*b$ 增长到 $*c$ 增长的幅度更小 (“从后”的规则)。

一、既然判断 P 中的元素是否可以增长需要与 Q 中的元素比较，那么直接用 Q 中最大的元素 $q_{max}(= \max\{x|x \in Q\})$ 与 P 中的元素去比较，则小于 q_{max} 的元素都是可以增长的；而大于 q_{max} 的元素，都是不可增长的。这样最大的小

于 q_{max} 的元素就是切点所在了，用 $p_{out}(= \max\{x|x \in P, x < q_{max}\})$ 表示。当然，如果不存在比 q_{max} 小的元素，即表示组合的字典序结束，无法再增长了。

二、找到切点，然后就是判断切点要增长到多少了。从上一步可知，切点右边的元素都是大于 q_{max} 的，所以切点需要的元素，只能在小一点的 Q 里了。在 Q 中找到最小的大于 p_{out} 的元素，即是切点需要的元素，用 $q_{in}(= \min\{x|x \in Q, x > p_{out}\})$ 表示。

三、在知道切点要从 p_{out} 增长到 q_{in} 之后，剩下的则是调整切点之后的元素了。由于是升序，所以我们需要大于 q_{in} 的数。 P 中 p_{out} 右边的元素有 $> q_{max} \geq q_{in}$ 的关系， Q 中 q_{in} 右边的元素有 $> q_{in}$ 的关系。所以应选出这两段中最小的元素来填充切点之后的尾巴，剩下的元素就充入 Q 了。这一步用特殊的归并算法可方便地完成。

这三步即可完成一次组合的字典序的枚举。

5 无重复元素的算法定义

现在可较正式的下定义了：

1. 用 Q 的最大元素 q_{max} 在 P 中找出最大的小于 q_{max} 的元素 p_{out} ；
2. 如果存在 p_{out} ，则
 - (a) 用 p_{out} 找出 Q 中最小的大于 p_{out} 的元素 q_{in} ，并用 q_{in} 替换 p_{out} ；
 - (b) 将 P 中原 p_{out} 右边的元素与 Q 中原 q_{in} 右边的元素取出，最小的元素填入原 p_{out} 右边，剩下的填入原 q_{in} 右边。

6 考虑重复元素

考虑重复元素的存在，需要对上面的进行一点改动，但有一点考虑是不变的，即“做最小的增长”。考虑到重复元素并不会改变大小关系，所以需要多加考虑的是在重复的元素中取哪一个位置的问题。

第一步选择 p_{out} 。由于 q_{max} 的值与位置无关，所以不变。但是用 q_{max} 来定位 p_{out} 时，就可能存在若干重复的 p_{out} 。按照“从后”的原则，选择最末的元素。如此，这一步就要改成“找出‘最末’的小于 q_{max} 的元素为 p_{out} ”。由于元素组合是升序的，所以“最末”亦即最大。

第二步用 q_{in} 替换 p_{out} ，亦需做改变。如果存在重复的 q_{in} ，则 q_{in} 应该是最前的元素，这样在第三步时就能得到最小的尾巴（显然，一个平稳的尾巴要小于一个上升的尾巴）。升序之中，最前即最小。

第三步由于位置确定，就不需要改变了。

下面即是可以在重复元素的定义，也是在最终算法中所使用的。

1. 用 Q 的最大元素 q_{max} 在 P 中找出最“末”的小于 q_{max} 的元素 p_{out} ；
2. 如果存在 p_{out} ，则

- (a) 用 p_{out} 找出 Q 中最“前”的大于 p_{out} 的元素 q_{in} ，并用 q_{in} 替换 p_{out} ；
- (b) 将 P 中原 p_{out} 右边的元素与 Q 中原 q_{in} 右边的元素取出，最小的元素填入原 p_{out} 右边，剩下的填入原 q_{in} 右边。

很抱歉，就这么处理了有重复元素的情况，缺乏有效的过渡（和证明），实在是不妥的。但是请考虑那句话：“增长一个最小的幅度。”

7 另一面，逆字典序

在写完逆字典序之后，我发现其实逆字典序很简单。发现了吗？在表1中列出的那些字典序的组合中，区间 P 的一列是正字典序，而区间 Q 的一列则是逆字典序。所以，逆字典序与正字典序是极为相似而且互补的，只是把选入的元素与选出的元素换个位置。因此在实现上，有正字典序一种即可，另一种变换一下参数即可达到。

逆字典序的定义可如下：

1. 用 P 的最大元素 p_{max} 在 Q 中找出最“末”的小于 p_{max} 的元素 q_{out} ；
2. 如果存在 q_{out} ，则
 - (a) 用 q_{out} 找出 P 中最“前”的大于 q_{out} 的元素 p_{in} ，并用 p_{in} 替换 q_{out} ；
 - (b) 将 Q 中原 q_{out} 右边的元素与 P 中原 p_{in} 右边的元素取出，最小的元素填入原 q_{out} 右边，剩下的填入原 p_{in} 右边。

8 实现

借助STL中丰富的算法，我的combination得以异常的简洁；不过我自己写了一个STL中所没有的归并的函数，比较奇特，但非常有用。

在比较元素的大小时，可以使用STL中的lower_bound和upper_bound来定位。前者在升序中找到最前的插入位置；后者则是找到最后的插入位置。稍加变化，两个函数分别可用来查找最后的小于某个值的元素和最前的大于某个值的元素。

而我自己写的twice_merge则可以用于将两个有序区间归并，然后将最小的元素放到前一区间，最大的元素放到后一区间。这就是第三步做的事。

下面分几步说明。

8.1 数据接口

元素用两个串接的区间表示，这样就用前文说过的($first, middle, last$)形式来表示。所以combination需要三个迭代器($first, middle, last$)的参数来表征元素区间。 $[first, middle)$ 表示当前组合中的元素，相当于 P ； $[middle, last)$ 表示当前选出的元素，相当于 Q 。

与STL的**permutation**类似，**combination**当输入是一轮组合的最后一组（这样输出便是下一轮字典序的第一组）时，返回**false**，表示一轮的结束（亦即新一轮的开始）；其它返回**true**，表示输出结果仍然是本轮的组合，没有结束。

同样的，以前缀**next**表示正字典序，以**prev**表示逆字典序。两个接口如下：

```
template <typename Iter>
bool next_combination (Iter first, Iter middle, Iter last);

template <typename Iter>
bool prev_combination (Iter first, Iter middle, Iter last);
```

8.2 最大最小：bound 之

在有序的区间里，可用STL中的**bound**算法来确定某个值的位置，以此来求出最大的小于它的元素，或最小的大于它的元素。

lower_bound(*first, last, x*)的返回值为[*first, last*)中第一个不小于*x*的元素的位置（不存在则为*last*）；则其前一位置即是最大的小于*x*的元素（或者不存在）。而**upper_bound**(*first, last, x*)的返回值为[*first, last*)中最小的大于*x*的元素的位置（不存在则为*last*）。

可见，两个**bound**函数可以完成 p_{out} 与 q_{in} 的选择。

lower_bound与**upper_bound**的接口：

```
template <typename Iter, typename T>
Iter lower_bound (Iter first, Iter last, const T& value);

template <typename Iter, typename T>
Iter upper_bound (Iter first, Iter last, const T& value);
```

8.3 归并，填充：twice_merge

如前，由[*first, middle*)与[*middle, last*)的升序排列的特性可知，($p_{out}, middle$)与($q_{in}, last$)也是升序的，并且所有比 q_{in} 大的元素都在这两个区间里。这样可以直接对两个区间归并后再进行填充，其中，较小的元素要填充到($p_{out}, middle$)中，较大的元素填充到($q_{in}, last$)中。

对于这个填充的动作，用一个特殊一点的算法来表示：

```
template <typename Iter>
void twice_merge (Iter first1, Iter last1,
                 Iter first2, Iter last2);
```

此函数将两个有序的空间归并，将其中最小的元素填充到[*first1, last1*)，剩下的元素填充到[*first2, last2*)。在此算法中，[*first1, last1*)对应($p_{out}, middle$)，[*first2, last2*)对应($q_{in}, last$)。

twice_merge 的代码如下:

```
template <typename Iter>
void twice_merge (Iter first1, Iter last1,
                 Iter first2, Iter last2)
{
    size_t len1 = distance(first1, last1);
    size_t len2 = distance(first2, last2);

    typedef iterator_traits<Iter>::value_type T;
    T *p = new T[len1+len2];
    merge (first1, last1, first2, last2, p);

    copy (p, p+len1, first1);
    copy (p+len1, p+len1+len2, first2);

    delete [] p;
}
```

一个更好的实现, 可以用不多于 $\min(len1, len2)$ 的空间实现。

8.4 完整的算法代码

好了, 现在到了完毕的时候了。这里先只给出next_combination 的实现代码; prev_combination 的实现与此是十分相似的。后面将用统一的代码实现next_combination 与prev_combination。

```
template <typename Iter>
bool next_combination (Iter first, Iter middle, Iter last)
{
    Iter pout = lower_bound (first, middle, *(last-1));
    if (pout == first) // max{Q} is less than min{P}
    {
        twice_merge (first, middle, middle, last);
        return false;
    }
    --pout; // just get the Pout

    Iter qin = upper_bound (middle, last, *pout);
    iter_swap (pout, qin); // swap, then fill
    twice_merge (pout+1, middle, qin+1, last);
    return true;
}
```

对于正字典序与逆字典序的组合算法，其实可以采用统一的实现方法，如下：

```
template <typename Iter>
bool combination (Iter first1 , Iter last1 ,
                 Iter first2 , Iter last2)
{
    Iter pout = lower_bound (first1 , last1 , *(last2 -1));
    if (pout == first1)      // max{Q} is less than min{P}
    {
        twice_merge (first1 , last1 , first2 , last2);
        return false;
    }
    --pout;                  // just get the Pout

    Iter qin = upper_bound (first2 , last2 , *pout);
    iter_swap (pout , qin);  // swap, then fill
    twice_merge (pout+1 , last1 , qin+1 , last2);
    return true;
}

template <typename Iter>
inline bool
next_combination (Iter first , Iter middle , Iter last)
{
    return combination (first , middle , middle , last);
}

template <typename Iter>
inline bool
next_combination (Iter first , Iter middle , Iter last)
{
    return combination (middle , last , first , middle);
}
```

9 辅助函数

本算法需要一些辅助函数来帮助建立“合法”的组合区间。因为不是任意的两个区间在一起，都是满足本算法的有效的组合。

辅助函数有两个，分别为**combination_init**与**combination_adjust**，前者用于初始化元素至一轮组合的字典序的开始或者结尾；后者初始化一个可能的组合（给定被选入的元素与选出的元素）至合法状态。另外，为了使元素有序，他们又使用了一个对区间排序的函数（简单的归并排序）。

combination_init 的接口如下:

```
template <typename Iter>
void
combination_init (Iter first , Iter middle, Iter last ,
                 bool min = true);
```

这个函数处理 $[first, last)$ 之间的元素。如果min为真,则初始化为最小的组合;否则,初始化为最大的组合。

combination_adjust 的接口如下:

```
template <typename Iter>
void
combination_adjust (Iter first , Iter middle, Iter last);
```

这个函数以 $[first, middle)$ 为选入元素,以 $[middle, last)$ 为选出元素,使组合合法化。

函数__combination_sort 为归并排序的函数,用于初始化区间至有序。

```
template <typename Iter>
void __combination_sort (Iter first , Iter last);
```

10 代码获取

请注意,这里给出的算法都是示意代码,严谨的程序代码查看正式的程序代码。代码可在网站<http://www.bxmy.org>获取。under GPL。

正式程序中使用的是统一正字典序与逆字典序的方法。在实际代码中,考虑了区间为空的情况,另外把twice_merge 的调用合并了。

代码中给出了另外两种形式的实现,一种为计数法,另一种为无重复元素的计数法。

参考

- libstdc++-3.x, ./include/c++/bits/stl_algo.h
- 排列组合的数学含义