第5章 组合数学基础

组合数学是现代数学的一个重要分支，研究的主要内容是有限、可数或离散对象。计算机科学的一个重要方向就是通过算法对离散数学问题进行加工和处理。随着计算机科学的发展，组合数学在计算机科学中的作用也日益凸显。排列和组合是组合数学中最基本的概念，本章主要介绍排列和组合生成的基础算法。

5.1 排列生成算法

将 n 个元素(或符号)按照确定的顺序进行重排，重排后的所有顺序称为全排列。图 5-1 给出了集合为{1,2,3,4}时的全排列示意图。

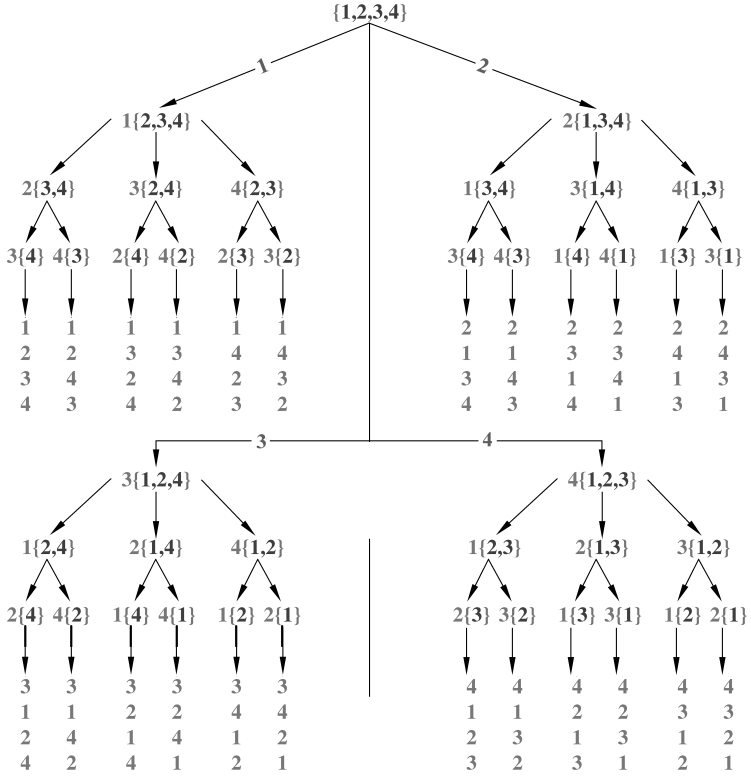

图 5-1 集合为{1,2,3,4}时的全排列示意图

全排列生成都遵循着原排列→映射规则→新排列的基本过程。全排列生成过程中，映射规则最为关键，不同映射规则决定了生成全排列的不同算法。常用的全排列生成算法包括序数生成法、字典序生成法、邻位互换法和轮转生成法等算法，本节只介绍序数生成法和字典序生成法两类。

5.1.1 序数生成法

基于序数的全排列生成算法，其核心思想是建立集合 S 的全排列与某一规则 R 所生成序列间的一一映射关系。建立映射时通常采用类似进制转换的方法，详细推导过程读者可参考组合数学相关书籍，本节只给出概要描述。

设 $p_1p_2\cdots p_n$ 为包含 n 个元素的集合 $S=\{s_1,s_2,\cdots,s_n\}$ 的一个排列，规则 R 定义如下。

① 对 $p_1p_2\cdots p_n$ 按值降序排列生成新的序列 $p'_1p'_2\cdots p'_n$。

② 规则 R 的生成序列为 $a_{n-1}a_{n-2}\cdots a_1$，与 $p'_1p'_2\cdots p'_n$ 的映射关系为

$a_{n-1}\leftrightarrow p'_1$，$a_{n-2}\leftrightarrow p'_2$，$\cdots$，$a_1\leftrightarrow p'_{n-1}$，其中 a_{n-i} 为 p'_i 在排列 $p_1p_2\cdots p_n$ 的逆序数。

逆序数是指排列中在值 k 右侧却比 k 小的元素的个数，如 4213 中 4 的逆序数为 3，2 的逆序数为 1。p'_n 值最小，没有逆序数，因而 R 的生成序列 $a_{n-1}a_{n-2}\cdots a_1$ 中没有关于 p'_n 的映射。

序数生成法生成全排列有以下两个关键点。

(1) 以阶乘为基的整数表示方法。

n 个数的全排列共有 $n!$ 个，用 $0\sim n!-1$ 表示。设 m 为 $0\sim n!-1$ 中的任意值，则 m 可表示为 $1!,2!,\cdots,(n-1)!$ 的线性组合，即 $m=a_{n-1}(n-1)!+a_{n-2}(n-2)!+\cdots+a_11!$，其中 $0\leqslant a_i\leqslant i$。$m$ 与唯一序列 $(a_{n-1}a_{n-2}\cdots a_i\cdots a_1)$ 一一对应。

(2) 逆序和排列一一对应。

因为排列的逆序和排列一一对应，所以排列的逆序就可以通过序列 $a_{n-1}a_{n-2}\cdots a_i\cdots a_1$ 唯一表示。

例如，当排列 $p_1p_2p_3p_4=4213$ 时，$p'_1p'_2p'_3=432$，此时 $a_3\leftrightarrow p'_1=4$，逆序数有 2，1 和 3，所以 $a_3=3$；$a_2\leftrightarrow p'_2=3$ 没有逆序数，所以 $a_3=0$；$a_1\leftrightarrow p'_3=2$，逆序数为 1，所以有 $a_1=1$。因此，排列 $p_1p_2p_3p_4=4213$ 对应的逆序序列为排列 $a_3a_2a_1=301$。

再例如，已知 $S=\{1,2,3,4\}$，逆序序列为 $a_3a_2a_1=301$，求对应的排列 $p_1p_2p_3p_4$。由 $S=\{1,2,3,4\}$ 可知，$a_3\leftrightarrow p'_1=4$。因为 $a_3=3$，可以确定在 4 右侧且小于 4 的元素有 3 个，所以 4 在排列的第 1 位；$a_2\leftrightarrow p'_2=3$ 且 $a_2=0$，可以确定在 3 右侧且小于 3 的元素有 0 个，所以 3 在排列的第 4 位；$a_1\leftrightarrow p'_3=2$ 且 $a_1=1$，可以确定在 2 右侧且小于 2 的元素有 1 个，所以 2 在排列的第 2 位；最后一位 1 排在第 3 位。表 5-1 给出了有 4 个元素的集合 $S=\{1,2,3,4\}$ 的全排列的映射。

表 5-1　集合 $S=\{1,2,3,4\}$的全排列的映射

序号	$a_3a_2a_1$	$p_1p_2p_3p_4$	序号	$a_3a_2a_1$	$p_1p_2p_3p_4$	序号	$a_3a_2a_1$	$p_1p_2p_3p_4$
0	000	1234	8	110	1342	16	220	3412
1	001	2134	9	111	2341	17	221	3421
2	010	1324	10	120	3142	18	300	4123
3	011	2314	11	121	3241	19	301	4213
4	020	3124	12	200	1423	20	310	4132
5	021	3214	13	201	2413	21	311	4231
6	100	1243	14	210	1432	22	320	4312
7	101	2143	15	211	2431	23	321	4321

序数法生成排列算法代码中 GenReverse()函数的功能是由整数生成以阶乘为基的逆序序列，整数与逆序序列的对应关系如表 5-1 所示。逆序序列递增 $a_1a_2\cdots a_{n-1}$的实际生成顺序与算法描述相反，输出时逆序即可。GenReverse()函数有 3 个参数，reverses[]是结果数组，下标从 1 开始，实际长度为 n；N 为待处理整数；len 为结果数组的长度。

GenPermutation()函数的功能是由逆序序列生成排列，生成方法可参考文献[7]对应章节的内容。GenReverse()函数有 3 个参数，permutation[]数组保存待生成的排列，下标从 1 开始，长度为 $n+1$；reverses[]保存已经生成的逆序数组，下标从 1 开始，实际长度为 n；len 为结果数组的长度。实现代码如下。

```
#include<iostream>
#include <cstdlib>
using namespace std;
void GenReverse(int reverses[], int N, int len)
{
    int i;
    for (i = 1; N > 0; i++)
    {
        reverses[i] = N % (i + 1);
        N = N / (i + 1);
    }
    while (i <= len - 1)
    {
        reverses[i++] = 0;
    }
}
void GenPermutation(int permutation[], int reverses[], int len)
{
    int i, j;
    for (i = 0; i <= len; i++)                    //将排列数组重置为 0
```

```
        permutation[i] = 0;
    for (i = len - 1; i >= 1; i--)
    {
//下标 j:从 p 的右(下标最大)侧向左(下标小)第 1 个未被占用的元素开始数,数 a[i]个数位
        j = len;
        while (true)
        {
            if (permutation[j] == 0)
            {
                reverses[i]--;
                if (reverses[i] < 0)
                    break;
            }
            j--;
        }
        permutation[j] = i + 1;                    //将该数位的值置为 i+1
    }
    for (i = 1; i <= len; i++)                     //最后一个元素位置填上 1
    {
        if (permutation[i] == 0)
            permutation[i] = 1;
    }
}
void DisplayResults(const char s[], int list[], int len, bool reverse)
{
    int i;
    cout << s << endl;
    if (!reverse)
    {
        for (i = 1; i < len; i++)
            cout << " " << list[i];
    }
    else
    {
        //逆序数组实际是按 R1R2...Rn-1 方式存放,需逆序输出
        for (i = len - 1; i >= 1; i--)
            cout << " " << list[i];
    }
    cout << endl;
}
int main()
{
    const int MAX_RECORDS = 10;                    //最多 10 个元素
    //下标从 1 开始:逆序数组有 9 个元素,排列数组有 10 个元素
```

```
    int rev[MAX_RECORDS], perm[MAX_RECORDS + 1];
    int i, n, fact;
    cout << "请输入排列的位数值:";
    cin >> n;
    fact = 1;
    for (i = 1; i <= n; i++)                    //求 n 的阶乘
        fact *= i;
    for (i = 0; i < fact; i++)                  //循环求出 p 数组
    {
        GenReverse(rev, i, n);                  //由整数生成以阶乘为基的表示结果
        DisplayResults("逆序", rev, n, 1);      //显示生成的逆序结果
        GenPermutation(perm, rev, n);           //根据逆序序列生成排列
        DisplayResults("排列", perm, n + 1, 0);
        cout << endl;
    }
    return 0;
}
```

5.1.2 字典序生成法

字典序生成法就是按照字典顺序规定了字符集中字符的先后关系，并以此为基础规定两个全排列的先后顺序是从左至右逐个比较相应的字符来确定。

设 n 个元素的集合 $S=\{s_1,s_2,\cdots,s_n\}$，其字典序的初始排列方案为 $P=p_1p_2\cdots p_n$，按照字典顺序得到 P 的下一个排列方案 P_{next}，称为 P 的一个置换。按照相同的转换方法重复 $n!-1$ 次就可以获得 n 个元素的全排列。从当前排列方案按照字典序获得下一字典序排列的置换方案表述如下。

(1) 找到最后一个正序。

找到最后一个正序末位可以表示为 $i=\max\{j\mid p_j<p_{j+1}\}$。从右至左(也可从左向右寻找，但从右向左寻找更易于理解)扫描 $P=p_1p_2\cdots p_n$ 的递减区间找到正序的末位 p_j，p_j 满足 $p_j<p_{j+1}$ 且 $p_{j+1}>p_{j+2}>\cdots>p_n$。

(2) 在递减区寻找大于 p_i 的最小元素 p_k。

在递减区间 $p_{i+1}p_{i+2}\cdots p_n$ 中从右至左寻找大于 p_i 的最小元素 p_k，即 $k=\max\{j\mid p_j>p_i\}$。对 p_i 右侧的递减序列而言，从右至左为递增，因此 k 是所有大于 p_i 的数字中序号最大者。

(3) 交换 p_i 与 p_k。

(4) 将原递减区升序排列。

例如，令 $S=\{1,2,3,4\}$，其某个字典序排列为 $P=3\ 4\ 2\ 1$，寻找下一个置换方案的过程如下。

(1) 最末正序为 $i=\max\{j\mid p_j<p_{j+1}\}=1$，即 $p_1=3$；

(2) 在递减区寻找大于 p_1 的最小元素的下标 $k=\max\{j\mid p_j>p_i\}=2$，$p_2=4$；

(3) 交换 p_1 和 p_2，交换后序列为 4 3 2 1；

(4) 升序排列原递减区间 4 3 2 1⇒4 1 2 3。

再例如,集合 $S=\{1,2,3,4,5,6,7,8,9\}$的某个字典序列 $p=1\ 4\ 6\ 2\ 9\ 5\ 8\ 7\ 3$,下一个字典序的置换过程为 1 4 6 2 9 5 8 7 3⇒1 4 6 2 9 7 8 5 3⇒1 4 6 2 9 7 3 5 8。

5.1.3 “火星人”问题

人类终于登上了火星,并且见到了神秘的火星人。人类和火星人都无法理解对方的语言,科学家发明了一种用数字交流的方法。首先,火星人把一个非常大的数字告诉人类科学家,科学家破解这个数字的含义后,再把一个很小的数字加到这个大数上面,把结果告诉火星人,作为人类的回答。

火星人用一种非常简单的方式来表示数字——掰手指。火星人只有一只手,但这只手上有成千上万的手指,这些手指排成一列,分别编号为 1,2,…。火星人的任意两根手指都能随意交换位置,他们就是通过这种方法计数的。

一个火星人用一个人类的手演示了如何用手指计数。如果把五根手指——拇指、食指、中指、无名指和小指分别编号为 1,2,3,4 和 5,当它们按正常顺序排列时,形成了 5 位数 12345,当你交换无名指和小指的位置时,会形成 5 位数 12354,当你把五个手指的顺序完全颠倒时,会形成 54321,在所有能够形成的 120 个 5 位数中,12345 最小,它表示 1;12354 第二小,它表示 2;54321 最大,它表示 120。表 5-2 展示了只有 3 根手指时能够形成的 6 个 3 位数和它们代表的数字。

表 5-2 手指数字与十进制数对应表

三进制数	123	132	213	231	312	321
数字	1	2	3	4	5	6

假如你有幸成为第一个和火星人交流的地球人。一个火星人会让你看他的手指,科学家会告诉你要加上去的很小的数。你的任务是把火星人用手指表示的数与科学家告诉你的数相加,并根据相加的结果改变火星人手指的排列顺序。

这是一道典型的字典序生成全排列问题。为此,添加了 SwapArrayElements()函数和 SelectionSort()函数作为辅助函数。SwapArrayElements()函数用于交换数组内两下标所对应的元素,SelectionSort()函数用于实现对给定区间进行选择排序。

SwapArrayElements()函数有 3 个参数,data[]为保存生成序列的数组,idx 和 idy 为待交换的两元素对应的下标,就是 p_i 和 p_k 对应的下标。SelectionSort()函数有 3 个参数,data[]为待排序数组,start 为排序的起始下标,len 为待排序长度。

按照字典序生成全排列的关键函数为 GetNextPermutation()。GetNextPermutation()在当前排列的基础上生成下一个置换,生成方法如前所述。GetNextPermutation()函数有两个参数,data[]为当前排列,count 为排列中元素的总数。实现代码如下。

```
#include<iostream>
#include <climits>
using namespace std;
//交换数组中下标为 idx 和 idy 的两个元素
```

```
void SwapArrayElements(int data[], int idx, int idy)
{
    int t = data[idx];
    data[idx] = data[idy];
    data[idy] = t;
}
//data[]为待排序数组,start 为排序的起始下标,len 为待排序长度
void SelectionSort(int data[], int start, int len)
{
    int i, j, temp;
    for (i = start; i < start + len; i++)
    {
        int min = i;
        for (j = i + 1; j < start + len; j++)       //遍历未排序的元素
            if (data[j] < data[min])                //保存目前最小值下标
                min = j;
        if (min != i)                               //若最小值不是当前位置则交换
        {
            temp = data[min];
            data[min] = data[i];
            data[i] = temp;
        }
    }
}
bool GetNextPermutation(int data[], int count)  //下标从 1 开始
{
    int i = count;
    //查找递减区间起始位置
    while (data[i] < data[i - 1])
        i--;
    i--;
    //在递减区间内找到与 data[i]最接近值的下标 position
    int min = INT_MAX, position = 0;
    for (int k = i + 1; k <= count; k++)
    {
        if (min > data[k] - data[i] && data[k] > data[i])
        {
            position = k;
            min = data[k] - data[i];
        }
        if (position == 0)
            return false;
    }
    //置换
```

```
    SwapArrayElements(data, i, position);
    //对原递减区域按升序排序
    int start = i + 1;
    SelectionSort(data ,start, count - start + 1 );
    return true;
}
int main()
{
    const int MAX_ELEMENTS = 40000;
    int elements[MAX_ELEMENTS];
    int count, perms;
    cin >> count;
    cin >> perms;
    for (int i = 1; i <= count; i++)
        cin >> elements[i];
    for (int i = 1; i <= perms; i++)
        if (!GetNextPermutation(elements, count))
            break;
    for (int i = 1; i <= count; i++)
        cout << elements[i] << " ";
    return 0;
}
```

运行结果如下。

```
9
1
1 4 6 2 9 5 8 7 3
1 4 6 2 9 7 3 5 8 请按任意键继续. . .
```

```
5
3
1 2 3 4 5
1 2 4 5 3 请按任意键继续. . .
```

5.2 组合生成算法

组合是组合数学中最基本的概念之一。从含有 n 个元素的集合 $S=\{s_1,s_2,\cdots,s_n\}$ 中任取 m 个作为一组(不考虑组内元素间的排列顺序),称为 S 的一个 m 组合,记作 C_n^m(也有 n 和 m 位置调换的表示方法)。从 n 个元素取出 m 个所构成的组合数量用式(5.1)表示。

$$C_n^m=\frac{n(n-1)(n-2)\cdots(n-m+1)}{m(m-1)(m-2)\cdots 1}=\frac{n!}{m!\ (n-m)!} \tag{5.1}$$

因为组合中不考虑元素间的排列次序,m 个元素的排列数为 $m!$,所以分母为 $m!$。组合具有以下两条性质:

(1) $C_n^m=C_n^{n-m}$;

(2) $C_n^m=C_{n-1}^m+C_{n-1}^{m-1}$。

性质(1)可以自己推导得出。性质(2)也可以理解为从 n 个元素取出 m 个,有如下两种情况:①不考虑第 n 个元素,只从前 $n-1$ 个元素中取 m 个;②选择第 n 个元素,再从

前 $n-1$ 个元素中取 $m-1$ 个。

5.2.1 基于字典序的组合生成算法

与排列相比，组合的生成要容易得多。表 5-3 给出从 $S=\{1,2,3,4,5,6\}$ 中任取 3 个元素的组合结果。

表 5-3 从 S 中任取 3 个元素的组合

序号	$c_1c_2c_3$	序号	$c_1c_2c_3$	序号	$c_1c_2c_3$
1	123	8	145	15	246
2	124	9	146	16	256
3	125	10	156	17	345
4	126	11	234	18	346
5	134	12	235	19	356
6	135	13	236	20	456
7	136	14	245		

观察表 5-3 中的每个组合 $c_1c_2c_3$，可以发现 $c_1c_2c_3$ 满足条件 $1\leqslant c_1<c_2<c_3\leqslant 6$，由此可以确定 c_1 的最大值为 4，c_2 的最大值为 5，c_3 的最大值为 6。若组合的各个数位之值都已经到达上限，则生成结束。否则，从右向左寻找第一个未达到上限的数 k，并将 $k+1$，$k+2$，…，$k+(r-j+1)$ 赋值给从该位开始到结尾的各个数位。例如，当组合 $c_1c_2c_3=126$ 时，第 3 位已经达到上限，从右向左未达到上限的第 1 个数值是第 2 位的 2，所以将 3(2+1)赋值给第 2 位，并将 4(2+2)赋值给第 3 位后结束。因此，$c_1c_2c_3=126$ 的下一个组合为 $c_1c_2c_3=134$。

从 n 个元素取出 m 个元素的解题思路：因为 m 个元素间不考虑排列，所以可以考虑使用一个具有 m 个元素的数组以升序方式来保存选择的每个元素，这就是组合的字典序生成法。

设集合 $S=\{s_1,s_2,\cdots,s_n\}$，按字典序生成其 m 组合。设 S 的一个 m 组合为 $a_1a_2\cdots a_m$，并且 $1\leqslant a_1<a_2<\cdots<a_n\leqslant n$。按字典序生成下一个组合的思路如下。

(1) 寻找最大下标 $i=\max\{j\mid a_j<n-m+j\}$。从右向左寻找第一个未达到上限的数，其下标即为所求；

(2) 进行两步处理：①$a_i\leftarrow a_i+1$；②$a_j\leftarrow a_{j-1}+1$，其中 $j=i+1,i+2,\cdots,r$。

NextCombination()函数是实现从当前组合生成下一组合的关键函数，参数 comb[]是保存当前组合的数组，下一组合生成后也存储于其中；n 为总元素个数；m 为构成组合元素的个数。为了处理方便，保存组合的下标及组合对应的数值均比实际值小 1，该值在输出组合时会进行补偿，以确保输出结果正确。有 3 个因素对于函数的处理逻辑有重要影响，分别是变量 i、ubound 和 ubound + i。变量 i 为组合最后一个元素的下标，ubound 为 n 与 m 的差值，ubound + i 为组合各位对应的上限。ubound + i 非常关键，随着 i 的减小，ubound + i 会变为对应位置的上限。以 C_5^3 为例，对应各数位的下标及其上限分别

为0(2),1(3),2(4)(为了处理方便,下标及上限均比实际值小1),表5-4给出了分析过程。

表5-4 下标及其限界对应表

N	m	i	ubound	ubound+i
5	3	2	2	4
		1		3
		0		2

1. 非递归生成法

使用NextCombination()函数生成下一个有效组合数的过程中,do循环从右向左找到第1个达到上限的位置,并将其前一位的数值加1。寻找上限时,先将当前位+1,即使超出上限也无所谓(输出时会调整)。例如,当前组合为125时,其实际值为014,此时$i=2$、ubound+$i=4$,处理过程如下。

(1) 循环内将最后一位增加,变为126(实际值为015)。

(2) while部分处理循环条件时,i值变为1,下一轮循环中ubound+i变为3。

(3) 新一轮循环中,循环体执行后,组合第2位值增加,变为136(实际值为025),此时第2位已经不大于ubound+i的值4(实际为3),循环结束。

if (comb[0] > ubound)分支结构的作用是判定整个组合生成过程是否结束。while循环用来处理组合后面数位上的值超出上限的情况。实现代码如下。

```
#include <iostream>
using namespace std;
bool NextCombination(int comb[], const int n, const int m)
{
    //所有元素值比实际输出值小1,输出时调整
    int i = m - 1;                                    //组合最后一个元素的下标
    //组合第1元素的上限
    const int ubound = n - m;                         //比实际输出值小1,输出时补偿1
    do
    {
        comb[i]++;
    } while (comb[i] > ubound + i && i--);
    //若第1位也已经到达上界,则组合生成完毕
    if (comb[0] > ubound)
        return false;
    //调整i位之后其余位:aj<-aj-1+1,
    while (++i < m)
        comb[i] = comb[i - 1] + 1;
    return true;
}
```