Построение перестановок

Условие задачи. Дано произвольное множество символов, построить все перестановки из его элементов.

Мы уже знаем, что хорошая идея не появляется на пустом месте. Необходимо четко и ясно представлять себе, как получаются перестановки, и уметь их строить, для чего нужны примеры, много примеров. Но конечно, чтобы не загромождать текст, приведем только один, по возможности удачный.

Таблица. Пример перестановок над множеством из 4-х букв

1	ABCD	7	DABC	13	CDAB	19	BCDA
2	ABDC	8	DACB	14	CDBA	20	BCAD
3	ADBC	9	DCAB	15	CBDA	21	BACD
4	ADCB	10	DCBA	16	CBAD	22	BADC
5	ACDB	11	DBCA	17	CABD	23	BDAC
6	ACBD	12	DBAC	18	CADB	24	BDCA

В таблице приведен пример множества перестановок, полученных из четырех элементов, роль которых играют четыре заглавные буквы латинского алфавита A, B, C, D. Этот пример может играть роль определения, из него в принципе видно, что такое перестановки. Для того чтобы закрепить понимание, заметим, что:

- все элементы множества присутствуют в каждой перестановке;

- перемена места двух элементов порождает новую перестановку. Иначе говоря, если в двух перестановках, хотя бы в одной позиции стоят разные элементы — это разные перестановки.

Наш пример построен очень удачно. Это, конечно, не случайно, пример специально подобран, что возможно, только если идея уже известна. Получается так, что показан не реальный процесс поиска идеи, а что-то искусственно подобранное. Но это не так. Просто, если демонстрировать все промежуточные шаги и метания, книга может стать безразмерной, а решение каждой задачи — хаотическим нагромождением не вполне понятных действий. Мы многое сокращаем для большей прозрачности. Впрочем, если вы не поленитесь и составите достаточно много примеров перестановок, то в конце концов даже интуитивно вы начнете строить их в каком-то порядке, хотя быть может и не таком, как продемонстрировано в таблице. Способность к упорядочению фундаментальна для нашего интеллекта.

Важное замечание

Вообще очень полезно, перед тем как непосредственно переходить к решению задачи, разобрать в деталях, что означает то или иное понятие. Нарисовать какие-то иллюстрирующие картинки, построить таблицы и диаграммы. Причем это полезно даже в том случае, если вы уверены в своем твердом знании определения. Но твердое знание определения процесса и детальное понимание хода процесса — это не одно и то же. Определение несет в себе минимум информации, а для того чтобы найти хорошее

решение, нужен информационный максимум. Грамотная картинка поможет не только более детально разобрать понятие, но и найти хорошую идею решения.

Итак, нам нужна идея. Прочитаем внимательно то, что написано о перестановках. В замеченных свойствах есть одна важная деталь. Сказано, что перемена места элементов порождает новую перестановку. Отсюда возникает мысль — можно построить одну первоначальную перестановку и затем придумать способ, построения новой перестановки из уже имеющейся. Этот способ должен быть таков, чтобы:

- пользуясь им, можно было получить все перестановки;
- перестановки не повторялись.

До сих пор наши рассуждения были последовательны, одно вытекало из другого, к сожалению так не получится до конца, всегда наступает момент, когда приходится положиться на интуицию. Хорошая идея — это всегда интуитивный прорыв, и с этим ничего не поделаешь. Посмотрите внимательно на таблицу, может быть у вас получится найти идею интуитивно.

Если же вы внимательно посмотрели, и у вас не получилось, то давайте вместе. Для начала попробуем придумать вообще любой способ получение новой перестановки из уже имеющейся. Кстати, наверное, это можно записать в виде правила.

Правило упрощения

Если вам необходимо организовать сложный процесс, но пока не получается, попробуйте организовать похожий, но простой. Может быть позже этот простой удастся усовершенствовать.

А вот простой способ получения перестановок: новая перестановка получается из уже имеющейся перестановкой элементов по кругу: первый на место второго, второй на место третьего … последний на место первого. Для четырех элементов нашего примера это будет выглядеть так: ABCD, DABC, CDAB, BCDA. Все полученные перестановки действительно разные, но если мы попробуем продолжить, то все последующие уже будут повторениями. Но то, что удалось получить несколько разных, вселяет надежду и можно попробовать метод улучшить. Немного подумаем. Нам не удастся получить новую перестановку вращая все четыре элемента, но можно вращать три. Возьмем первую перестановку из полученных ранее и прокрутим три последних элемента. Получится вот что: ABCD, ADBC, ACDB. Как видите три полученные перестановки не повторяются. Если мы повторим точно такую же операцию со всеми четырмя, то получим 4*3 = 12 различных перестановок.

Дальше проще. Теперь мы возьмем по очереди все 12 перестановок и для каждой из них повращаем уже только две последние буквы. Для примера из первой получим: ABCD, ABDC. Соответственно 12 перестановок превращаются в 24. Вот и все. Теперь можно сказать, что идея решения есть, но выглядит она еще достаточно туманно и, наверное, записать ее в виде алгоритма на данном шаге будет сложно. Поэтому попробуем выразить ее не в виде точного алгоритма, а виде более строгого описания. Помните, в *главе 1* было сказано, что процесс разработки алгоритма можно представить, в виде отдельных шагов на каждом из которых выполняется уточнение уже понятного текста.

Уточнение. Для того чтобы получить все перестановки, берем любую из них за исходную, затем запускаем циклический процесс, в ходе которого на каждом шагу получается новая перестановка прокруткой по кругу элементов текущей. Если в исходной перестановке N элементов, то перед тем как в очередной раз прокрутить N элементов, мы должны N–1 раз прокрутитьN–1 элемент. В общем случае, перед тем как в очередной раз прокрутить k элементов, надо k–1 раз прокрутить k–1 элемент. Отчет элементов можно вести слева направо.

Возможно, что-то по-прежнему непонятно, возможно, все кажется понятным, но тем не менее, что-то упущено. Попробуем по нашему описанию отработать еще один пример, попроще. Это нам даст или уверенность, что все ясно, или выявит какую-нибудь скрытую проблему. Возьмем исходное множество только из трех символов: A, B, C. И построим все возможные перестановки:

1. Исходная перестановка — ABC.

2. Прокрутим два последних и получим новую перестановку — ACB.

3. Мы говорили, что два элемента надо прокрутить два раза. Поэтому выполняем прокрутку еще раз и получаем ABC. Мы действовали по алгоритму и получили повтор. Это плохо, но тем не менее построим все, что получится. Нам нужна информация.

ABC	CAB	BCA
ACB	CBA	BAC
ABC	CAB	BCA

Три перестановки действительно повторились, но похоже в этих повторениях есть какая-то закономерность. Если мы ее выявим, то сможем лишние перестановки отбрасывать. Конечно, следует заметить, что появление лишних перестановок означает какую-то потерю качества алгоритма (расчет лишних величин — это плохо) и может быть следует подумать о другом алгоритме, свободном от этого недостатка, но прежде чем отказываться от идеи лучше проанализировать, так ли это плохо. Может быть указанным недостатком можно пренебречь.

Примечание

Достаточно часто приходится мириться с недостатком в алгоритме. Машинное время, конечно, важный фактор, но если машина выиграет несколько минут, а вам надо потратить на разработку более эффективного алгоритма несколько часов, то стоит ли его разрабатывать? Кроме того, всегда следует спросить себя, а смогу ли я сделать лучше? И надо ли делать лучше? Что мы выиграем, разработав более эффективный алгоритм? Это вопросы не ленивого человека. Просто представьте себе ситуацию коммерческой разработки. Вам как программисту дали задание. Естественно, кроме содержательной части в этом задании будет указано, за какой срок задание необходимо выполнить. Это в свою очередь означает, что свобода эксперимента самым серьезным образом ограничена.

Конечно, это не означает, что следует выбрать первый попавшийся алгоритм и на нем остановиться. Необходимо искать компромисс между временем разработки и эффективностью алгоритма. Пока у вас мало опыта, нужно меньше жалеть времени, потом с опытом вы сможете за то же время находить более эффективные решения.

А сейчас займемся главной проблемой нашего будущего алгоритма. Предстоит выяснить, когда появляются лишние перестановки, как от них избавиться и по возможности оценить их количество. Анализ последнего примера показывает, что лишние перестановки появляются по одной на каждую прокрутку трех элементов. Причина появления лишней перестановки в том, что мы фактически прокручиваем два элемента три раза.

Видимо на каждую прокрутку 3 элементов приходится одна лишняя из 2, на каждую прокрутку из 4 — одна лишняя из 3. Далее, прокруток из 3 элементов — 3, следовательно прокрутки из 3 порождают 3 лишние из 2 элементов. Аналогично 4 прокрутки из 4 порождают 4 лишние из 3 элементов. Таким образом, при N элементах мы имеем N*(N–1)*…*3 лишних при том, что неповторяющихся перестановок ровно N!. Это означает, что в общей массе перестановок лишних почти столько же, сколько и не повторяющихся. Плохо это или нет зависит о того, сколько элементов в исходном множестве, на котором строятся перестановки. Надо только заметить, что скорость роста факториала столь велика, а числа факториалов достигают астрономических величин так быстро, что увеличение их еще на треть вряд ли серьезно ухудшает дело — оно и так достаточно плохо. Поэтому можно сделать вывод о целесообразности нашего алгоритма, если количество элементов в исходном множестве не слишком велико.

Все принципиальные моменты построения алгоритма уже обговорены, но для сложных алгоритмов обсуждение только принципиальных вопросов часто бывает недостаточным. Зачастую недоговоренные технические детали создают не меньшие проблемы, чем принципиальные идеи. Поэтому займемся мелочами.

Заметим, что для каждой прокрутки из 4 элементов необходимо построить все перестановки из 3. А для каждой прокрутки из 3, необходимо построить все перестановки из 2 и т. п. То есть можно сказать, что прокрутка из N элементов определяется всеми прокрутками из N–1 элемента. Определения такого вида, когда конструкция или значение определяются значениями или конструкциями той же природы, но отличающимися количественно называются *рекуррентными*. А если величина определена рекуррентно, то для ее получения можно построить рекурсивную программу. Это очень серьезное утверждение, давайте оформим его в виде правила.

Правило построения рекурсии

Задача, если в ее математическом решении применяются рекуррентные определения, допускает, рекурсивное решение, и скорее всего это решение будет наиболее простым и естественным.

Возможно, из того, что было сказано ранее не совсем понятно, что такое рекуррентное определение. Для пояснения приведем два примера математических величин и для каждого два определения: рекуррентное и нерекуррентное .

Таблица. Примеры рекуррентных и не рекуррентных определений

Название величины	Нерекуррентное определение	Рекуррентное определение
Факториал	N!=1*2*…*N	N!=N*(N-1)!; 1!=1
Арифметическая прогрессия	$A_N=A_0+d*(N-1)$	$A_N=A_{N-1}+d$; A_0=Число

Если вспомнить, что рекурсивная процедура (или функция) — это такая процедура (или функция), которая в процессе своей работы вызывает сама себя, то становится понятным откуда между рекуррентным определением и рекурсивной процедурой такая хорошая взаимосвязь. У них один принцип работы. Рекуррентное определение не дает значение величины сразу, оно только говорит, какую величину той же природы надо подсчитать, чтобы вычислить данную. Рекурсивная процедура (функция) поступает так же, она не вычисляет величину, она только определяет, с какими значениями запустить процедуру еще раз, чтобы данный вызов мог закончить свою работу. Зачастую, для построения рекурсивного модуля достаточно записать рекуррентное определение. К примеру, тело функции, рекурсивно вычисляющей факториал, будет выглядеть так:

If n>1 then factorial:=n*factorial(n-1) else factorial:=1;

Обратите внимание на еще одну общую черту. И в рекуррентном определении и в рекурсивном модуле часто необходимо выделить тривиальный случай (то есть простой) на котором расчеты завершаются. А теперь вернемся к нашей задаче, запишем алгоритм и текст программы.

Главный алгоритм:

```
Ввести исходное множество — массив длиной N элементов
Вызвать процедуру РЕКУРСИЯ с аргументом N
```

Процедура РЕКУРСИЯ — входное значение N:

```
Делать N раз
  Построить перестановку на N элементах
  Если N>1 То вызвать процедуру РЕКУРСИЯ с аргументом N-1
```

Листинг

```
program example;
  uses crt;
  var
   n_big,m,i:integer;
   a:array[1..10] of char;
   c:char;
procedure printing;
  var
   j:integer;
begin
  m:=m+1;write(m,')  ');
  for j:=1 to n_big do write(a[j],' ');
  writeln;
c:=readkey;
end;
procedure recurs(n:integer);
  var
```

```
   i,j:integer;
begin
  for i:=1 to n do
    begin
      c:=a[n];
      for j:=n downto 1 do a[j]:=a[j-1];
      a[1]:=c;
      if i<n then printing;
      if n>1 then recurs(n-1);
    end;
end;
begin
  clrscr;m:=0;readln(n_big);
  for i:=1 to n_big do readln(a[i]);
  clrscr;printing;
  recurs(n_big);
end.
```

Ранее было сказано, что наличие рекуррентного определения означает возможность получения хорошего рекурсивного решения. Это, однако, не означает, что возможно только рекурсивное решение. Для каждой задачи можно придумать как рекурсивное, так и не рекурсивное решение. Решение без рекурсии зачастую более понятно и требует меньше ресурсов. Программирующим на Pascal под DOS, кроме того, необходимо помнить, что рекурсия реализуется посредством стековой памяти, размер которой невелик. А теперь попробуем найти нерекурсивное решение.

Мы построим его на той же идее, что и рекурсивное. В этом нет ничего неожиданного. Рекурсия — это форма организации программы точно так же, как рекуррентные формулы — это форма организации вычислительного процесса. Поэтому, вполне можно ожидать, что и для рекурсии и для обычного решения окажется возможным использовать одну и ту же идею, лишь немного иначе оформленную.

Напомним, что идею кратко можно записать так: алгоритм представляет собой цикл прокруток элементов массива разной длины. Рекурсивное определение перестановки длины N требует построения всех перестановок меньшей длины. Переход к нерекурсивному алгоритму должен это как-то учесть. Заметим, что рекурсивная процедура занимается счетом прокруток. Это делает оператор if n>1 then recurs(n-1);. Он вместе с циклом обеспечивает многократное построение меньших перестановок. Отсюда возникает идея счета количества перестановок определенной длины.

Общая идея. Заведем счетчики для перестановок. Счетчик с номером 2 будет считать количество перестановок длины 2, счетчик с номером 3 будет считать перестановки длины 3 и т. д. Тогда ядро алгоритма — это цикл, в теле которого осуществляется поиск счетчика, чье значение меньше его номера. Если, к примеру, будет обнаружено, что счетчик с номером 5 имеет значение 3, то это означает, что есть возможность выполнить прокрутку длины 5.

Проверка значений счетчиков должна выполняться с меньшего номера. Это необходимо для того, чтобы обеспечить условие "Новую перестановку длины N можно строить только тогда, когда построены все перестановки меньшей длины", об этом условии мы уже говорили. Как только построена новая перестановка длины N, опять

появляется необходимость в построении всех перестановок меньшей длины, для чего соответствующие счетчики потребуется обнулять.

Счетчик с номером 1 нам не нужен, так как переставлять последний элемент сам с собой нет смысла.

Общая идея сформулирована, важные технические детали описаны, можно записать алгоритм и текст программы.

```
Ввести исходное множество
Обнулить счетчики прокруток
Пока значение счетчика с номером N меньше N, делать
  Найти счетчик с наименьшим НОМЕРОМ, такой что его значение
     меньше его НОМЕРА.
  Выполнить прокрутку длиной равной НОМЕРУ
  Увеличить значение найденного счетчика на 1
  Если значение счетчика меньше его НОМЕРА то
    Распечатать перестановку.
    Обнулить все счетчики с меньшими номерами
```

Листинг

```
program example;
uses crt;
var
 a:array[1..20] of char;
 c:array[1..20] of integer;
 w,n,i,k:integer;
 d:char;
procedure printing;
var
 k:integer;
begin
 w:=w+1;
 write('w=',w,' ');
 for k:=1 to n do write(a[k],' ');
 writeln;
end;
begin
 clrscr;
 readln(n);
 for i:=1 to n do
  begin
   readln(a[i]);
   c[i]:=0;
  end;
 clrscr;
 w:=0;
 printing;
 while c[n]<n do
  begin
   i:=2;
   while c[i]=i do i:=i+1;
   c[i]:=c[i]+1;
```

```
   d:=a[1];
   for k:=1 to i-1 do a[k]:=a[k+1];
   a[i]:=d;
   if c[i]<i then
    begin
     printing;
     for k:=2 to i-1 do c[k]:=0;
    end;
  end;
end.
```

В заключение.

Обратите внимание, что при перестройке рекурсивного решения в нерекурсивное пришлось ввести дополнительный массив. Этот массив на самом деле совсем не нов. Он был и в рекурсивном варианте. Стек, используемый рекурсией, и есть дополнительный массив. Разница в том, что за управлением стеком нам следить не приходится, вот почему рекурсивное решение выглядит проще.