Иван Гаврюшин

Гуманитарные основы комбинаторных алгоритмов

Издательские решения
По лицензии Ridero
2023

УДК 004
ББК 32.973
Г12

Шрифты предоставлены компанией «ПараТайп»

Гаврюшин Иван

Г12 Гуманитарные основы комбинаторных алгоритмов / Иван Гаврюшин. — [б. м.] : Издательские решения, 2023. — 66 с.
ISBN 978-5-0060-0460-3

Книга о гуманитарных основах комбинаторных алгоритмов представляет собой сборник статей и заметок, опубликованных в Интернете автором в период с 2015 по 2022 год. Издание предназначено для программистов с гуманитарным образованием, интересующихся философскими истоками информационных наук, также для широкого круга читателей.

УДК 004
ББК 32.973

12+ В соответствии с ФЗ от 29.12.2010 №436-ФЗ

ISBN 978-5-0060-0460-3

Гуманитарные основы комбинаторных алгоритмов. Сборник статей и заметок (информационные науки, разработка и программирование).

И. Н. Гаврюшин

2023 год

1) Два в шестой степени. Или один очевидный факт о Книге Перемен

Есть такой памятник древнекитайской письменности, называемый

«Книга Перемен». Подробно о нём можно почитать в Википедии или ином энциклопедическом источнике. Я лишь напомню, что этот текст считается гадательным. Для тех, кто не любит читать энциклопедии: книга состоит из гексаграмм и комментариев к ним. Каждая гексаграмма состоит из палочек двух видов: сплошной «___» и прерывистой «__ __». Палочки символизируют собой начала «Инь» и «Ян». Палочек в каждой гексаграмме очевидно шесть (если вдруг забыли, что означает латинский корень гекс). Об этом памятнике писали многие учёные, неучёные, о нём писали эзотерики, искусствоведы, математики, философы, филологи, обычные блогеры и многое множество интересующихся китайской культурой людей.

Немного истории

Саму книгу на русский язык перевёл востоковед Щуцкий Юлиан Константинович. Книгу изучал знаменитый философ Лейбниц. И, может быть, именно эта книга во многом натолкнула его на разработку двоичной системы счисления. Но на этом исторический обзор я вынужден закончить, так как в общем и целом об этом памятнике и его изучении написано много.

О комбинаторике гексаграмм

Вернёмся непосредственно к самим гексаграммам, число коих в книге 64. Вооружённый микроскопом информатики разум сразу заметит, что число 64 — это степень двойки, иначе говоря двойка в шестой степени. То есть самих гексаграмм в книге 2 в 6 степени.**«И какой из этого вывод?»** — спросит любознательный читатель. Казалось бы, никакого, но вот только гекскаграммы, символизирующие различные состояния, не повторяются, то есть все 64 гекскаграммы уникальны. И из это следует один примечательный факт: гексаграммы Книги Перемен формально могут рассматриваться в качестве комбинаторного объекта, а именно как размещения с повторением из двух по шесть. Однако компьютерного алгоритма, порождающего последовательности гексаграмм я пока не встречал. Скорее всего, его и не существует. И возможен ли какой-то строгий алгоритм в этом случае?! Я не проверял. При первом рассмотрении последовательности гексаграмм видно, что изменения в них происходят не по чёткому простому математическому принципу, а по каким-то иным законам. В качестве примера возьмём только первые гексаграммы так, как они приведены в оригинальном памятнике: в первой гексаграмме все чёрточки полные, во второй сразу разделённые, в третьей появляются сразу две полные; в четвёртой полные меняют порядок следования, в пятой полных чёрточек уже три, в шестой их четыре и т.д., и т. д. Мы наблюдаем «нарастание и убывание двух начал», их игру во времени.

Заключение

В одной из статей мной была обнаружена замечательная вещь: оказывается существует другой порядок следования гексаграмм, отличный от канонического. Об этом можно посмотреть здесь: https:// www.synologia.ru/, статья называется «К вопросу об истоках последовательности чисел-символов». Но в связи с этим возникает закономерный вопрос: **так сколько же возможно порядков следования гекскаграмм?!**

Сведущий в вычислении факториалов математик даст довольно точный, но по-своему сногсшибательный ответ. Но будет ли такой ответ верным применительно к самой жизни, состояния которой в общем-то, как мне кажется, и описывает Книга Перемен.

2) Комбинаторные свойства русского текста

«Не комбинатор — не внидет»

Главным фактором, побудившим автора взяться за данную заметку стали занятия переборными алгоритмами, эксперименты с выведением комбинаторных алгоритмов без опоры на учебники и пособия, их программирование.

Результаты практической работы можно найти в сетевом хранилище:

https://github.com/dcc0?tab=repositories

Заметки с примерами алгоритмизации можно посмотреть на habr.com. Для детального изучения вопроса мне понадобилось поработать с 8 классическими алгоритмами перебора множеств и одним достаточно новым алгоритмом порождения суперперестановок (англ. superpermutations). Речь идёт прежде всего об алгоритмах: 1) перестановок (англ. permutations) размещений (англ. arrangements), 3) сочетаний (англ. combinations) 4) разбиений (англ. partitions). Каждый из данных четырёх алгоритмов подразумевает существование этого же алгоритма, но с повторением, именно поэтому алгоритмов получается 8. Ознакомление с данными алгоритмами и комбинаторикой в целом необходимо для выявления и понимания структурной или комбинаторной сложности текстов на русском языке. Читателю предлагается вспомнить, что представляет собой следующие принципы взаимодействия с объектами или операции над ними, именуемые: перестановка, разбиение, сочетание, размещение, композиция.

Я исхожу из следующего посыла: содержание и глубина содержания определяют сложность текста. Органичная сложность русского текста задаёт сложность его математической, комбинаторной структуры, его красоту. Мысль о содержании предопределяющим органическую красоту

выведена мной на основе рассуждений Николая Лосского в работе «Мир как осуществление красоты». В своей работе Николай Лосский критически относится к мысли академика В. Виноградова об отношении «формы и содержания». В его позиции прослеживаются следы идеализма и субъективизма по вопросам эстетики и восприятия. Однако я переношу ранее высказанную мысль с эстетических вопросов на сложность русского текста в структурном аспекте. Таким образом основная идея данной заметки звучит так: органичная сложность содержания задаёт не только красоту, но и определяют структурную сложность текста. В этой связи глубокое понимание содержания текста задаёт предпосылку к глубокому понимаю математики текста, а также развивает способность видеть в тексте то, что не является в тексте математическим. Вопрос о самом определении сочетания органичная сложность не рассматривается.

Слово и фраза имеют комбинаторную структуру и могут изучаться в качестве независимых дефиниций. Слово является вложенной структурной единицей в комбинаторную структуру предложения или фразы. Это «комбинаторика в комбинаторике». Фраза более общая дефиниция относительно слова. Незначительные погрешности во вложенной структуре или изменения могут практически не влиять на создаваемый фразой образ. Слово и фраза — это структура внутри другой структуры; для слова минимальной единицей является буква, для фразы — слово. В отличие от числовой комбинаторики комбинаторика слов естественного языка обнаруживает некоторые законы, которые, говоря математическим языком, можно назвать не в полной мере детерминированными. Например,

В русском слове за согласной буквой часто следует гласная. Помимо комбинаторики букв в слове выделяется также комбинаторика слогов, создаваемая по определенным правилам, в основном правилам фонетики, иногда с отступлением от этих правил, что в математическом аспекте не даёт возможности эти правила детерминировать в полной мере.

Для понимания комбинаторной сложности русских текстов можно попытаться сравнить законы построения предложения в русском и английском языках. Как известно, в английском языке прямой порядок слов, что ограничивает, например, возможность перестановки слов в предложении, хотя поэтическая речь в некоторой степени допускает незначительные перестановки слов. Русский текст лишён этого ограничения почти полностью и допускает практически любой порядок слов в предложении, что в разной степени сказывается на оттенках смысла и значительно дифференцирует русский текст в стилевом плане. Рассмотрение русского текста через призму комбинаторики позволяет изучать его как многомерную связанную структуру, однако при подобном рассмотрении и особенно попытках перенесения комбинаторного взгляда с текста на речь, следует помнить, что некоторые законы и правила письменного текста для устного текста — речи — далеко не всегда работают.

Завершить данную заметку хотел бы эпилогом с рифмой:

Комбинаторика! Как прекрасна ведь она! Но все же разума игра!

Post Scriptum

Для примера возьмём две фразы. «Am I like a god?» и «Am I like a dog?»

Пример того, как можно связать атомизм Демокрита и комбинаторику. Мы переставили буквы в одном слове, осуществили

комбинаторный прием — перестановку и получили новый смысл вопроса. Более того, оба вопроса приобрели для нас религиозно-философский смысл. Для самой комбинаторики наше действие, может быть, не имеет смысла, оно похоже на механическое, но оно и результат, который оно даёт, имеет смысл для нас, оно имеет смысл для нас как для волевого, разумного, чувствующего субъекта, иначе говоря для состояния самого нашего духа.

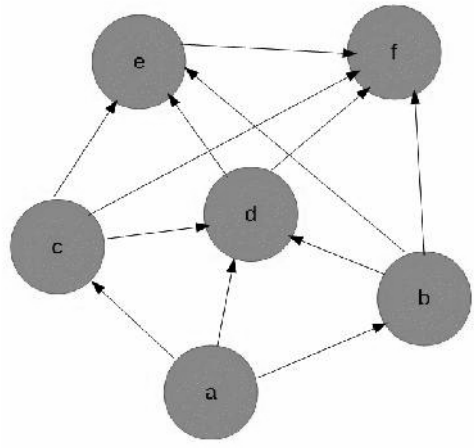
Игра с комбинаторными алгоритмами

3) Путешествие из Москвы в Казань через Санкт-Петербург или процесс разработки алгоритма поиска всех путей

Данный материал публикуется с расчетом на начинающих программистов и неспециалистов…

Однажды вечером после чтения книжек о путешествиях, — кажется, это были знаменитое «Путешествие из Петербурга в Москву» Радищева и «Тарантасъ» Владимира Соллогуба — я сел смотреть лекцию об алгоритме Дейкстры. Смотрел, рисовал что-то на бумажке и нарисовал ориентированный граф. После некоторых размышлений мне стало интересно, как бы я реализовал алгоритм поиска всех путей из одной начальной точки (a) в какую-то другую единственную конечную точку (f) на ориентированном графе. Я уже было начал читать об алгоритмах поиска в глубину и ширину (https://studfiles.net/preview/5920852/page:3/), но мне подумалось, что интереснее было бы попробовать «изобрести» алгоритм заново, часто ведь при таком подходе можно получить интересную модификацию уже известного алгоритма. Заодно я поставил себе несколько условий: 1) не использовать литературу; 2) использовать нерекурсивный подход; 3) хранить ребра в отдельных массивах, без вложенности. Далее постараюсь описать процесс поиска алгоритма и его реализации, как обычно на PHP.

Сам граф получился такой:



В общем: на входе ориентированный граф с шестью вершинами, задача найти все пути из a в f без рекурсии и с минимальными затратами средств.

Ребра хранятся в нескольких массивах, имя массива — вершина:

```
$a=array ('b','c','d');
$b=array ('d','e','f');
$c=array ('d','e','f');
$d=array ('e','f');

$e=array ('f');
```

Чтобы получить первый путь, я решил пройтись по нулевым индексам каждого массива и склеить их в одну строку x (в этой переменной на каждом этапе будет хранится найденный путь). Но как это сделать с минимальными затратами?! Мне показалось, что самым простым вариантом будет ввести еще один массив — инициализирующий.

В массиве int все элементы, которые есть в графе в обратном порядке.

$int=array ('f','e','d','c','b','a');

Тогда получить первый путь очень просто, достаточно пройтись циклом по всем массивам, добавлять в x новое значение с помощью конкатенации, и на каждом этапе использовать элемент из предыдущего массива в качестве указателя на следующий массив.

Этот стиль немного напоминает bash, но код выглядит довольно понятным:

```
$x='a';
$z=$a [0];
while (1) {
$x.=$z;
$z=$ {$z} [0];
if ($z == 'f') {$x.=$z; break;}

}
```

И так, мы получили первый путь x=abdef.

Можем вывести его на экран и заняться непосредственно самим алгоритмом, так как все, что было выше, — это только подготовительная часть. По идее от нее можно было бы избавиться, но я ее оставляю и публикую, чтобы был лучше понятен ход мысли.

Выводим на экран первый путь и запускаем
первую функцию.

```
print $x;
print '<br>»;

search_el ($x,$a,$b,$c,$d,$e);
```

Сам алгоритм фактически сводится к двум циклам,
которые вынесены в отдельные функции.
Первая функция принимает полученный ранее
первый путь x. Далее в цикле осуществляется
обход x справа налево. Мы ищем два элемента,
один из которых будет работать в качестве
указателя на массив, другой (правый, тут только
стоит помнить, что массив перевернут)
в качестве указателя на элемент массива.
С помощью array_search найдем ключ элемента
и проверим, есть ли что-нибудь в данном
массиве после него. Если есть, то заменим
элемент на найденный, но перед этим отрежем
хвост (для этого нужен substr). Замену можно
организовать и по другому:

```
function search_el ($x, $a, $b, $c, $d, $e)
{

$j = strlen ($x);
while ($j!= 0)
{
$j — ;
if (isset ($ {$x [$j — 1]}))

$key = array_search ($x [$j], $ {$x [$j — 1]}); if ($ {$x [$j — 1]}
[$key +1]!= «»)
{
$x = substr ($x, 0, $j);
```

```
$x.= $ {$x [$j — 1]} [$key +1];
new_way_search ($x, $a, $b, $c, $d, $e); break;
}
}
}
```

Условие с isset нужно, чтобы интерпретатор не выбрасывал предупреждение. К самому алгоритму оно отношения не имеет. Если никаких элементов в массивах найдено не было, то алгоритм завершится, но если все-таки чудо свершилось, то переходим в новую функцию, суть которой крайне проста — дописать хвост к x, вывести на экран и… «дорисовать восьмерку» или петлю — вернуться в функцию, из которой мы пришли, но уже с новым значением x:

```
function new_way_search ($x, $a, $b, $c, $d, $e)
{
$z = $x [strlen ($x) — 1];
$z = $ {$z} [0];

while (1)
{
$x.= $z;
if ($x [strlen ($x) — 1] == 'f') break;

if ($z == 'f')
{
$x.= $z;
break;

}

$z = $ {$z} [0];
}
```

```
echo $x;
echo '<br />»;
search_el ($x, $a, $b, $c, $d, $e);

}
```

Результат работы алгоритма для графа, что на рисунке выше:

```
abdef
abdf
abef
abf
acdef
acdf
acef
acf
adef
adf
```

Дополнение

— качестве дополнения приведу описание полученного алгоритма более кратко: ребра ориентированного графа выписаны в отдельные массивы в порядке возрастания. Т.е. вершины графа и рёбра упорядочены. Это обязательное условие. До начала алгоритма находим первый путь, который с учетом первого условия будет с наименьшими именами вершин. Способ нахождения не особенно важен.

Описание для проверки на бумаге

На входе первый найденный путь x=abdef:

— Двигаемся справа налево по массиву x, выделяем два соседних элемента (кроме последнего) abd [e] f, левый (d) исполь-

зуем в качестве указателя на массив с вершиной, правый (e) в качестве указателя на элемент этого массива.

Ищем элемент в d после e, если он есть, убираем в x справа от e все элементы. Получаем в x=abde. Заменяем правый элемент (e) на найденный элемент.

— Дописываем (вторым циклом) правую часть от элемента (или индекса правого элемента), который был заменен и до последнего элемента (f). В этом цикле требуется брать всегда массивы с 0 индексом, так как массивы упорядочены по условию. В данном случае мы сразу получили в правой части последний элемент x=abdf, поэтому второй цикл сработает вхолостую.

— После формирования правой части возвращаемся к обходу массива справа налево.

Отсутствие элементов в первой вершине (массив a) — условие выхода из алгоритма.

Тот же код без функций и рекурсии, первый путь в x задан:

```
<?php

//Массивы ребер
$a=array ('b','c','d');
$b=array ('d','e','f');
$c=array ('d','e','f');
$d=array ('e');
$e=array ('f');

//Первый путь
$x='abdef';
print $x;
print '<br>»;

$j=strlen ($x);

while ($j!=0) {
```

```
//ищем новый элемент
$key = array_search ($x [$j], $ {$x [$j-1]});

//Если нашли, уберем правую часть и заменим
    элемент if ($ {$x [$j-1]} [$key+1]!= «») {
$x=substr ($x, 0, $j);
$x.= $ {$x [$j-1]} [$key+1];
$z=$x [strlen ($x) -1];

$z=$ {$z} [0];

//Собираем новый путь x с помощью нулевых индексов

while (1) {
//Если последний элемент равен f, то выйдем
    из цикла

if ($x [strlen ($x) -1] ==’f’) {$x.=$z; break;} $x.=$z;
$z=$ {$z} [0];
}
//Напечатаем
$j=strlen ($x);
echo $x;
echo '<br>»;
}
$j — ;

}

?>
```

Вариант с массивом

```
<?php
error_reporting (0);
$a=array (’b’,’c’,’d’);
$b=array (’d’,’e’,’f’);
```

```
$c=array ('d','e','f');
$d=array ('e');
$e=array ('f');
$x=array ('a','b','d','e','f');
print_r ($x);
print "\n»;
$j=count ($x) -1;
while ($j!=0) {
$key = array_search ($x [$j], $ {$x [$j-1]}); if ($ {$x
    [$j-1]} [$key+1]!= «») {
$x=array_slice ($x, 0, $j);
array_push ($x, $ {$x [$j-1]} [$key+1]);
$z=$x [count ($x) -1];
$z=$ {$z} [0];

while (1) {

if ($x [count ($x) -1] =='f') {array_push ($x, $z); $z=$
    {$z} [0];

array_push ($x, $z); break;}
}

$j=count ($x);
print_r ($x);
echo "\n»;
}
$j — ;
}
?>
```

О методологии вместо заключения

В итоге для разработки подобных алгоритмов получился довольно простой метод, который может быть полезен в работе с динамическими массивами. Общая его суть — проведение

подготовительных действий перед запуском основного алгоритма, для упрощения реализации. Это также должно делать алгоритм более прозрачным и понятным, что в свою очередь в дальнейшем должно способствовать его оптимизации и упрощению. В данном случае методика разбивается на три подготовительных шага.

1) Определение наличия порядка в данных. Упорядочивание, если необходимо.

2) Введение инициализирующего (вектора) массива на упорядоченных данных.

3) Получение начального пути на основе предыдущего шага, смысл которого также состоит в упрощении основного алгоритма. В данном случае начальный путь также должен строиться с учетом порядка данных так, чтобы не получилось пропуска какого-либо пути.

Если на вершинах графа нет порядка, то может понадобиться дополнительный шаг, переопределение названий вершин, фактически построение изоморфного графа и создание массива соответствий (например, между реальными названиями городов и буквами алфавита). Для других случаев алгоритмизации вынесение исходного пути (вектора) за пределы циклов позаимствовано мной из моих прошлых статей о порождении комбинаторных объектов: перестановок, разбиений/ композиций, сочетаний, размещений. Если говорить о конкретных рабочих реализациях, то конечно, стоит внимательно изучить возможности того или иного языка по работе с динамическими данными. В данной ситуации использование «переменных переменных» для определения значения одной переменной в качестве названия для другой является лишь способом демонстрации корректности самого алгоритма. Какие существуют риски при использовании данного подхода в рабочих условиях, автору неизвестно.

4) Нерекурсивный алгоритм генерации всех разбиений и композиций целого числа

Я снова решил поиграться с алгоритмизацией и написал вот эту статейку. Хотел даже отправить в какой-нибудь рецензируемый журнал, но оценить тривиальность/нетривиальность данного материала я не в состоянии, так как программирование всего лишь мое хобби и то эпизодическое, поэтому, как обычно, заранее прошу прощения, если все окажется слишком простым и до боли знакомым.

Итак, плоды усилий долгих...

Нерекурсивный или итеративный алгоритм генерации всех разбиений целого числа в лексикографическом порядке, когда все элементы выстроены в порядке убывания, является альтернативным; в Интернете представлено несколько способов порождения данных комбинаторных объектов, однако, как это справедливо и относительно других комбинаторных алгоритмов, реализации сводятся к двум типам — нерекурсивному и рекурсивному. Чаще можно встретить реализации, которые не учитывают порядок вывода объектов или осуществляют вывод по принципу дробления числа.

Приведенная ниже реализация работает по обратному принципу: исходное число изначально разбито на единицы, алгоритм работает до тех пор, пока число в нулевом индексе массива не станет равным сумме исходного числа. Особенностью данного алгоритма является то, что он крайне прост для понимания, однако это не лишает его некоторый специфики:

— Первый объект просто выводится на экран в самом начале, таким образом, он вынесен за пределы циклов, фактически является инициализирующим;

— Существует несколько способов реализации переноса единицы, которые могут, как упростить код, так и сделать его более запутанным;

— Данная нерекурсивная реализация может служить наглядным примером для объяснения генерации комбинаторных объектов на нескольких процессорах, после незначительной модификации. Для разделения генерации на процессоры достаточно: а) определить элемент по его номеру и сделать инициализирующим; б) определить момент для остановки работы. Например, если известно число объектов, генерируемых на одном процессоре, то достаточно ввести еще одну инкрементируемую переменную в верхний цикл и изменить условие выхода из самого верхнего цикла по достижении требуемого количества.

Код на языке PHP приведен только для демонстрации корректности алгоритма и может содержать лишние языковые средства, которые добавляют реализации избыточности.

Описание алгоритма

Дано: исходный массив в виде единиц —
A (1,1,1,1,1).

Шаги

— Если получили сумму (в случае реализации ниже, если нулевой индекс равен сумме числа), тогда остановка алгоритма.

— Двигаясь по массиву слева направо, искать в массиве A первый минимальный элемент — x,
последний элемент не учитывается (не участвует в поиске минимального).

— Перенести единицу из конца (последнего элемента) в найденный минимальный элемент x

(равносильно увеличению x на единицу и уменьшению на единицу последнего элемента).

— Если в массиве A есть ноль — 0, то удалить последний элемент.

— Разложить сумму всех элементов после измененного элемента — x — на единицы.

Пример

A= (1,1,1,1,1)

2,1,1,1
2,2,1
3,1,1

```
<?php
/*Генерация всех разбиений. Generate all
    partitions.*/ $a = array (
1,
1,
1,
1,
1,
1,
1
);
$n = count ($a);
while (1)
{
/*Печать и выход. Print end exit.*/
print_r ($a);

if ($a [0] == $n) break;
```

```
/*Элемент в нулевом индексе нашего
    динамического массива на текущий момент.
First element of our dynamic array*/

$first_elem = $a [0];

/*Размер массива на текущий момент. Length of an
    array*/ $c = count ($a) — 1;
$i = 0;
while ($i!= count ($a) — 1)
{

/*Найдем элемент меньше первого. Here we
    search min.
element.*/
if ($a [$i] <$first_elem)
{
$first_elem = $a [$i];
$min_elem = $i;
}
$i++;
}

if (empty ($min_elem)) $min_elem = 0;

/*Перенос элемента «1». Here we transfer «1». */
    $a [$min_elem] += 1;
$a [$c] -= 1;
```

/*Обрежем массив и найдем сумму его элементов. We cut the array

— and count the sum of elements.*/ array_splice ($a, $min_elem +1); $array_sum = array_sum ($a);

/*Добавим в массив единицы заново с учетом суммы.

Here we add 1 (fill) to the array

```
(taking into account the sum).*/
for ($j = 0; $j!= $n — $array_sum; $j++) $a [] = 1;

/*Обнулим переменную. Unset min_elem.*/ unset
($min_elem);
}

?>
```

Дополнение

Операция с поиском и удалением 0 в массиве лишняя (так как используется array_splice, а затем новое заполнение массива), как правильно было замечено в комментарии участником dev26

Выводы

Хотел бы в конце поделиться одним наблюдением, я очень долго пытался понять, почему одни алгоритмы понятны сразу и легки для кодирования, а другие заставляют мучиться… и мучиться порой долго. Должен отметить, что этот алгоритм у меня получилось закодировать почти сразу, но только после того, как я получил однозначно понятное описание каждого шага. И тут есть важный момент, понять алгоритм и описать — задачи одна другой не легче. Однако, в алгоритмизации и составлении описания, особенно важным оказывается то, какими глаголами описываются действия в алгоритме — это (субъективно) в конечном счете может влиять и на конечную реализацию.

Литература

— Donald E. Knuth. The Art of Programming. Vol. 4. 2008.
— Dennis Ritchie and Brian Kernighan. The C Programming Language. 1978.

— Aleksandr Shen. Algorithms and Programming: Problems and Solutions.

— ru.wikipedia.org/wiki/Разбиение_числа
— en.wikipedia.org/wiki/De_Arte_Combinatoria

P.S.: несмотря на приведенный список литературы, алгоритм пришлось выводить заново.

Еще одно дополнение: вынос первого элемента из циклов при данном подходе имеет силу как для генерации разбиений, так и для генерации сочетаний, перестановок. В принципе данный подход (хоть и несколько избыточный при реализациях) вполне обобщается для генерации других комбинаторных объектов.

И еще одно дополнение: алгоритм разбиений в соединении с алгоритмом перестановок позволяет генерировать и все композиции числа. Идея простая: для каждого разбиения вызывается функция генерации всех перестановок для этого разбиения.

```
<?php

/*Генерация всех разбиений. Generate all partitions.*/

$a = array (
1,

1,

1,

1,

1);
```

```
$n = count ($a);

$h=0;

while (1)

{
/*Печать всех перестановок и условие выхода.

Permutations and exit.*/

permute ($a, $h);

if ($a [0] == $n) break;

/*Элемент в нулевом индексе нашего динамического

массива на текущий момент.

First element of our dynamic array*/

$first_elem = $a [0];

/*Размер массива на текущий момент. Length of an array*/

$c = count ($a) — 1;

$i = 0;

while ($i!= count ($a) — 1)

{
/*Найдем элемент меньше первого. Here we search min.

element.*/
```

```
if ($a [$i] <$first_elem)

{
$first_elem = $a [$i];

$min_elem = $i;}


$i++;}


if (empty ($min_elem)) $min_elem = 0;

/*Перенос элемента «1». Here we transfer «1». */

$a [$min_elem] += 1;

$a [$c] -= 1;

/*Обрежем массив и найдем сумму его элементов. We cut the
array

* and count the sum of elements.*/

array_splice ($a, $min_elem +1);

$array_sum = array_sum ($a);

/*Добавим в массив единицы заново с учетом суммы.

Here we add 1 (fill) to the array

(taking into account the sum).*/
```

```
for ($j = 0; $j!= $n — $array_sum; $j++) $a [] = 1;

/*Обнулим переменную. Unset min_elem.*/

unset ($min_elem);}


/*Функция перестановок. Permutations function.*/

function permute ($b,&$h)

{
/*Дублируем массив и перевернем его. Это нужно для выхода

* из алгоритма.

* Here we make a copy and reverse our array. It is necessary to

* stop the algorithm.*/

$a = $b;

$b = array_reverse ($b);

while (1)

{
/*Печать и условие выхода. Here we print and exit.*/

print_r ($a);

print "\n»;

if ($a == $b) break;
```

```
/*Ищем слдующую перестановку.

* Here we search next permutation.*/



$i = 1;

while ($a [$i]> = $a [$i — 1])

{
$i++;}


$j = 0;

while ($a [$j] <= $a [$i])

{
$j++;}


/*Обмен. Here we change.*/

$c = $a [$j];

$a [$j] = $a [$i];

$a [$i] = $c;

/*Обернем хвост. Tail reverse.*/

$c = $a;

$z = 0;
```

```
for ($i-= 1; $i> — 1; $i — ) $a [$z++] = $c [$i];}}
    ?>
```

Вопрос читателям: при чтении описания данного алгоритма получается ли у Вас вывести его на листке бумаги?

5) Размышления об алгоритмах и методах. Представление полного алгоритма порождения сочетаний + размещений с повторением

Эта статья содержит ряд наблюдений, касающихся проблем алгоритмизации, минимизации ошибок, понимания и изучения чужого

кода, а также рассуждения о полном представлении алгоритмов и небольшой эксперимент.

Что я понимаю под полным описанием алгоритма и зачем это может быть нужно?

Без открытия Америки не обойтись: полный алгоритм — это та последовательность действий над объектами, которая осуществляется ручкой на бумаге. А полное описание — это каждый шаг, описанный на естественном языке, возможно, с применением дополнительных математических обозначений, но только в качестве вспомогательных. Таким образом, полное описание — это не формула и даже еще не псевдокод и, тем более, не блок-схема. Из этого очевидного тезиса следует такая же очень простая, но важная вещь — понимание реализации даже достаточно простого алгоритма может быть затруднено его сокращением или краткостью формулы. В связи с этим мне очень понравилось мнение Херба Уилфа о том, что «формула — на самом деле алгоритм», перепечатанное в данной статье Игорем Паком. Возможно, представление одних и тех же алгоритмов в виде разных формул и наблюдение за собственным восприятием этих представлений может пролить свет

на то, почему одни алгоритмы мы понимаем лучше, а другие хуже.

Я несколько раз замечал, как часто алгоритмы переписываются с языка на язык чисто механически. Вероятно, это может быть хорошо в коммерческой разработке, когда нет особенно времени вникать в каждый шаг, а лишь требуется быстро получить результат на нужном язык, но для учебных целей такой метод не очень подходит.

Понаблюдав за собой, я обратил внимание, что бывают ситуации, когда смотришь на чужой код и вроде бы понимаешь алгоритм, знаешь язык, на котором он реализован, но не видишь всех шагов. Автор продумал для себя алгоритм, максимально сократил и таким образом фактически скрыл пошаговую реализацию для других программистов, внимание которых зачастую больше обращено на средства реализации, а все усилия направлены на поиск аналогов этих средств, необходимых для перекодирования на известный им язык.

Таким образом, реализация и краткое формальное описание алгоритма порой выглядит как шифр, сходу понятный только автору. Что касается описания алгоритмов на естественном языке, то, наверное, разгадка сложности порой кроется в самом языке автора, так как при описании каждый опирается на свое знание и понимание терминологии и использует тот набор и сочетание речевых средств, которыми привык

оперировать. Вывод, который я сделал для себя в результате изучения некоторых алгоритмов и наблюдения за переводом с языка на язык, заключается в том, что восприятие формул, алгоритмов может быть темой для целой серии очень серьезных исследований с привлечением наработок из совершенно разных областей человеческой деятельности.

Вопрос: нужны ли реализации полных алгоритмов?!

Безусловно, реализации, например, полных комбинаторных алгоритмов могут работать в разы медленнее и могут быть более запутанными и некрасивыми, чем их сокращенные аналоги. На мой взгляд, подобные пошаговые реализации могут служить орудием своего рода реверс-инжиниринга, — когда нужно получить не механически переписанный с языка на язык код, а понимание всех шагов, что, кстати, в дальнейшем может послужить путём и к понимаю всех сокращений и витиеватостей в более быстрых и кратких аналогах. Модификация представления алгоритма сочетаний без повторений.

Не секрет, что при выведении на бумаге многие комбинаторные алгоритмы кажутся достаточно простыми. Например, алгоритм порождения сочетаний без повторений (или комбинаций без повторений — различаются хотя бы одним элементом) может быть выведен практически сходу без особенного осмысления, фактически интуитивно. Я, правда, при выписывании на листке всех сочетаний для n=5 по k=3 пару раз совершил одну и ту же ошибку, проигнорировав одно из чисел и получил неверный результат; это привело меня к мысли, что представление алгоритма на бумаге стоит осуществлять более системно, с максимальной визуализацией, с большим числом проверок при переходе от одного шага к другому, иначе то, что может быть реализовано за несколько минут, может быть растянуто на несколько часов или даже дней.

Далее мой эксперимент — представление полного алгоритма перебора сочетаний

Кратко алгоритм генерации сочетаний сформулирован так: «... в текущем сочетании найдём самый правый элемент, не достигший ещё своего наибольшего значения; тогда увеличим его на единицу, а всем последующим элементам присвоим наименьшие значения».

Источник: http://e-maxx.ru/algo/generating_combinations

Мне не очень понятно ни описание, ни реализация, поэтому я решил все немножко усложнить и добавить несколько дополнительных шагов, чтобы таким образом компенсировать свою невнимательность к числам и сделать описание более подробным.

Описание

На входе массив A=123456 (для примера), отсортированный по возрастанию; N=6 (кол-во символов в массиве); допустим, K=3,.

До входа в цикл входной массив предлагается разбить на два — B и C, B хранит значения от 1 элемента (включительно) и до K (включительно) и в C все остальное, т. е. B [123] и C [456].

— Во внешнем цикле осуществляется перебор элемента на позиции K в B, т. е. B [K], поиск элемента большего на единицу в массиве C и обмен. Вывод на экран.

— Далее условие — если B [K] равно N — последний элемент массива, то запускается цикл поиска элемента слева от K, для которого в массиве C есть больший на единицу. Если дошли до 0 индекса и таких элементов нет, то алгоритм завершается.

— Если же элемент в C найден, то ищется его индекс, чтобы в дальнейшем уменьшить элемент в C на единицу, а элемент в B увеличивается на единицу. Затем массив B разбивается на два (можно без этого, см. сокр. вариант); до текущего элемента и после. Все, что после текущего элемента, переносится в массив C.

Массив B увеличивается до индекса K по возрастанию, а из массива C удаляются лишние элементы. Завершается вложенный цикл. Операции повторяются заново. Использование

этого алгоритма оказывается более затратным по времени, но хранение дополнительного массива позволяет избежать ошибок, связанных с вниманием и делает понятным более краткие реализации. В результате после разложения алгоритма на большее число шагов, у меня получилось практически сразу его закодировать.

Однако, должен предупредить, что реализация не на бумаге, а на языке, например, на PHP может показаться очень запутанной. Но, тем не менее, формально алгоритм работает правильно.

Полный нерекурсивный алгоритм порождения сочетаний (combinations) без повторений. PHP

```
<?php
$a=array (1,2,3,4,5);
$k=3;
$n=5;
$c=array_splice ($a, $k);
$b=array_splice ($a, 0, $k);
$j=$k-1;
print_r ($b);

while (1) {
if (in_array ($b [$j] +1, $c)) {
$m=array_search ($b [$j] +1,$c);
if ($m!==false) $c [$m] -=1;
$b [$j] =$b [$j] +1;
print_r ($b);

}

if ($b [$k-1] ==$n) {
$i=$k-1;
while ($i> = 0) {
if ($i == 0 &&!in_array ($b [$i] +1, $c)) {break 2;
```

```
}
if (in_array ($b [$i] +1, $c)) {
$m=array_search ($b [$i] +1,$c);
if ($m!==false) $c [$m] =$c [$m] -1;
$b [$i] =$b [$i] +1;
$f=array_splice ($b, $i+1);
$b=array_splice ($b, 0, $i+1);

$c=array_merge ($c,$f);

$g=$i;
while ($g!= $k-1) {
$b [$g+1] =$b [$g] +1;
$g++;
}
$c=array_diff ($c,$b);
print_r ($b);
break;
}
$i — ;
}

}

}
?>
```

Сокращённый вариант с комментариями

```
<?php
$a=array (1,2,3,4,5);
$k=3;
$n=5;
$c=array_splice ($a, $k);
$b=array_splice ($a, 0, $k);
$j=$k-1;
```

```
//Вывод
function printt ($b,$k) {

$z=0;
while ($z <$k) print $b [$z++].»»;
print "\n»;
}

printt ($b,$k);

while (1) {
//Увеличение на позиции K до N
$m=array_search ($b [$j] +1,$c);
if ($m!==false) {
$c [$m] -=1;
$b [$j] =$b [$j] +1;

printt ($b,$k);

}
if ($b [$k-1] ==$n) {
$i=$k-1;
//Просмотр массива справа налево
while ($i> = 0) {
//Условие выхода

if ($i == 0 && $b [$i] == $n-$k+1) break 2; //Поиск
    элемента для увеличения
$m=array_search ($b [$i] +1,$c);
if ($m!==false) {
$c [$m] =$c [$m] -1;

$b [$i] =$b [$i] +1;
                                                    $g=$i;

//Сортировка массива B по возрастанию while ($g!= $k-1) {
```

```
array_unshift ($c, $b [$g+1]);
$b [$g+1] =$b [$g] +1;
$g++;
}
//Удаление повторяющихся значений из C
$c=array_diff ($c,$b);
printt ($b,$k);
break;
}
$i – ;

}

}

}

?>
```

Добавление

Интересно, что данный алгоритм довольно легко модифицируется в алгоритм порождения сочетаний с повторениями. Для этого достаточно по-другому задать массивы C и B, после удаления повторяющихся значений, на каждом проходе добавлять в C элемент под номером N. Вместо сортировки массива B по возрастанию, достаточно заполнить массив B после элемента K одним и тем же единожды увеличенным элементом.

Полный нерекурсивный алгоритм порождения сочетаний (combinations) с повторениями. PHP

```
<?php
$k=3;
$n=5;
$c=array (2,3,4,5);
```

```
$b=array (1,1,1);
$j=$k-1;
//Вывод
function printt ($b,$k) {

$z=0;

while ($z <$k) print $b [$z++].»»;
print "\n»;
}
printt ($b,$k);

while (1) {
//Увеличение на позиции K до N

if (array_search ($b [$j] +1,$c)!==false) {$b [$j] =$b [$j] +1;

printt ($b,$k);
}

if ($b [$k-1] ==$n) {
$i=$k-1;
//Просмотр массива справа налево
while ($i> = 0) {
//Условие выхода
if ($i == 0 && $b [$i] == $n) break 2; //Поиск
    элемента для увеличения
$m=array_search ($b [$i] +1,$c);
if ($m!==false) {
$c [$m] =$c [$m] -1;

$b [$i] =$b [$i] +1;
                                                    $g=$i;

//Сортировка массива B
```

```
        while ($g!= $k-1) {
        array_unshift ($c, $b [$g+1]);
        $b [$g+1] =$b [$i];
        $g++;
        }

    //Удаление повторяющихся значений из С
$c=array_diff ($c,$b); printt ($b,$k); break;
    }
$i – ;
    }}}
    ?>
```

Данный алгоритм можно использовать и для генерации размещений без повторений или с повторениями. Для генерации всех размещений с повторениями генерируем все сочетания и для каждого сочетания все перестановки. Модифицированный для этой задачи алгоритм перестановок приведен из предыдущей статьи:

Алгоритм генерации всех размещений с повторениями. PHP

```
<?php
set_time_limit (0);

//Функция генерации всех перестановок для разбиения
function perm ($b) {

    $a=array_reverse ($b);
    while (1) {
    print_r ($a);
    print '<br/>»;
    if ($a ==$b) break;
    $i=1;
    while ($a [$i]> = $a [$i-1]) {
```

```
$i++;
}
$j=0;
while ($a [$j] <= $a [$i]) {
$j++;
}
$c=$a [$j];
$a [$j] =$a [$i];
$a [$i] =$c;

$a=array_merge (array_reverse (array_slice ($a, 0,
    $i)),array_slice ($a, $i));}
}

//Генерируем все сочетания с повторением //
    установим к и n
$k=5;
$n=5;
//Установим массив
$c=array (1,2,3,4,5);
//установим массив b число единиц, в котором
    равно k $b=array (1,1,1,1,1);
$j=$k-1;
//Вывод
function printt ($b,$k) {

//На каждое сочетание генерируем все
    перестановки perm ($b);

}
printt ($b,$k);

while (1) {
//Увеличение на позиции K до N

if (array_search ($b [$j] +1,$c)!==false) {$b [$j] =$b [$j] +1;
```

```
printt ($b,$k);
}

if ($b [$k-1] ==$n) {
$i=$k-1;
//Просмотр массива справа налево
while ($i> = 0) {
//Условие выхода
if ($i == 0 && $b [$i] == $n) break 2; //Поиск
    элемента для увеличения
$m=array_search ($b [$i] +1,$c);
if ($m!==false) {
$c [$m] =$c [$m] -1;

$b [$i] =$b [$i] +1;
                                                    $g=$i;

//Сортировка массива B


while ($g!= $k-1) {
array_unshift ($c, $b [$g+1]);
$b [$g+1] =$b [$i];
$g++;}

    //Удаление повторяющихся значений из C
    $c=array_diff ($c,$b); printt ($b,$k); array_unshift
    ($c, $n); break;}

    $i – ;}}}
?>
```

Лирико-практическое дополнение

Уже после написания этой статьи я оказался в библиотеке им. Ленина с томиком «Трудов по не математике» (т. I)

В. А. Успенского, который в главе о Витгенштейне оставил такие строки, цитируя его самого: «Деятельность, деятельность и еще раз деятельность — вот кредо Витгенштейна. Для него процесс важнее результата. «Я открываю не

результат, а тот путь, которым он достигается.»». В.А.Успенский (Витгенштейн).

Post Scriptum

Свою прошлую статью о порождении сочетаний я опубликовал преждевременно, не заметив сразу несколько ошибок, поэтому извиняюсь за свою поспешность. Реализации нерекурсивного и рекурсивного алгоритмов порождения сочетаний на разных языках были обнаружены мной на сайте rosettacode.org (https://rosettacode.org/wiki/Combinations). Там же есть реализации алгоритма из книги Липского.

6) Программирование глазами (и руками) гуманитария. Личный опыт. Немного философии

У каждого, наверное, есть какая-то своя излюбленная тема в IT, помимо повседневной практической работы, приносящей порой лишь хлеб насущный. Может быть, кто-то в свободное время делает мультфильмы или работает над какой-нибудь игрой, может быть, участвует в разработке какого-нибудь социального проекта или просто изучает что-то новое без особенной на то практической нужды, одним словом — делает что-то для себя, так сказать, для души.

Мой скромной опыт программирования привел меня к мысли, что этот род деятельности может быть ценным сам по себе и не иметь конкретных практических целей. Безусловно программирование может приносить творческое удовлетворение и, может быть, кому-то своеобразное эстетическое наслаждение; в конце концов, в любой сфере люди жаждут видеть собствен-

ную гармонию. Я, когда-то давно, когда еще едва был знаком с программированием и даже плохо понимал значение слова функция применительно к написанию кода, пошутил, сказав, что программирование это искусство. Это была просто брошенная фраза за дружеским чайным столом. Кто-то подыграл мне, спросив, как я это могу доказать. Я ответил что-то вроде: «не зря же Дональд Кнут свою антологию назвал искусством программирования». А ведь он знает толк в этом.

В то время я относился к программированию, как к сухой трудно понятной мне науке, считая её смесью лингвистики и математики. Конечно, тогда я уже имел представление об HTML, но коды программ я видел только издалека и предпочитал держаться от всего этого подальше.

Компьютер — это печатная машинка, энциклопедия, переводчик, иногда игрушка — вот, что такое компьютер для гуманитария, лет так 10—15 назад. Конечно, сюда еще можно включить работу с почтой, может быть, еще пару вещей, но основное назначение компьютера для гуманитария — быть печатной машинкой.

Однако все меняется, когда появляется потребность сделать сайт. Я не знаю, сколько гуманитариев трудится в сфере информационных технологий и сколько из них занимается программированием, но я точно уверен, — нашего брата к написанию кода приводит именно WEB. По крайне мере так было еще совсем недавно. Одного молодого филолога-php-программиста из Сибири я недавно встретил на форуме dapf.ru, и он такой, скорее всего, не единственный. Если копнуть глубже, то можно вспомнить, что создатель языка Perl Ларри Уолл, — у которого 27 сентября день рождения, — лингвист по образованию. Но вернусь к теме искусства.

Своё отношение к уже множество раз упомянутой деятельности в этой заметке я изменил после того, как чисто умозри-

тельно попытался провести параллели между написанием кода и игрой в шахматы (или решением шахматных задач), а также теми эффектами, которые они приносят субъекту действия.

На мой взгляд, общего очень много. К программированию вполне возможно относиться как к своего рода игре, в которой есть место и хитрости, и психологии, и присущей только этой сфере магии. Возможно,

На примерах этой магии и говорят опытные программисты, когда рассказывают о том, как получили в наследство код, который почти невозможно поддерживать, но он каким-то чудом работает?!

На мой взгляд, **отношение к программированию как к игре,**

вероятно, может помочь преодолеть ряд психологических барьеров тем людям, которые считают написание кода уделом выпускников и студентов спецвузов. Новому поколению, правда, уже, наверное, не понять того, как наши родители еще лет 20 назад боялись нажать лишний раз не то сочетание клавиш. Теперь многое доступно, нет особенных страхов повредить домашний стационарный ПК, ибо у кого-то в запасе еще есть ноутбук, может быть, два, а под подушкой спрятан планшет. Одним словом, программировать можно не стесняясь…

Поскольку существенные этапы развития программирования прошли мимо меня, а мне всегда хотелось немножко заглянуть в эту сферу — отчасти из любопытства, отчасти в силу некоторых историко-научных интересов, — то я решил, пользуясь свободным временем, немножечко почитать учебник по языку Си Денниса Ритчи и что-нибудь закодировать,

что-нибудь потестировать и что-нибудь с чем-нибудь сравнить, чтобы было веселее. Простите, что не буду оригинальным, так как я решил воспроизвести — с использованием стандарт-

ных библиотек Си — тот самый алгоритм перестановок, о котором уже писал ранее.

Код получился очень громоздким, так как я старался побольше функций реализовать самостоятельно. Собственно, написание данного кода и было для меня своего рода игрой, так сказать, наслаждением для разума.

Должен добавить, что я бы не стал тревожить сообщество своим «гуманитарным кодом», а просто опубликовал бы эту заметку без кода и вообще упоминания каких-либо алгоритмов, если бы не один довольно интересный момент: мой длиннющий код с достаточным количеством функций и циклов после компиляции на Linux-системе оказался крайне шустрым. Я даже полез по старым ссылкам, чтобы найти наиболее быстрые реализации на других языках. По данной ссылке (https://stackoverflow.com/questions/3846123/generating-permutations-using-bash), в самом низу, рекурсивный алгоритм перестановок на awk, который помечен автором как наиболее быстрый.

Я решил сравнить скорость со своим учебным примером и, честно говоря, результаты меня удивили. Для n = 10 awk выдал вот такой результат: real time 1m16.770s (на всякий случай, машина: AMD Phenom T1100 6x)

В итоге: я допил холодный чай, который простоял на столе 2 часа и нажал Enter, так я запустил свой только что откомпилированный код.

Терминал выдал мне: real time 0m13.411s

Си, амиго, — это фантастика!

Код Си. Полный нерекурсивный алгоритм порождения всех перестановок в лексикографическом порядке (если читать справа налево.

```
#include <stdio. h>
#include <string. h>

#include <stdlib. h>

//This reverse x. Переворачиваем x
char revstring (char * x) {

int i = strlen (x);
int k=0;
char c;

while (i> k) {
i — ;
c=x [k];
x [k] =x [i];
x [i] =c;
k++;

}
}

//This cut x
char subb (char * x, int i) {
x [i] ='\0»;
}
//This cut y
char subb2 (char * y, int i) {
int k = 0;
while (k!= strlen (y) +1) {
y [k] =y [i];
i++;
k++;
}
}
```

```
//It gets an argumet like 1234 or abcd. All symbols
    must be uniqe int main (int argc, char *argv []) {
if (argc <2) {

printf («Enter an argument. Example 1234»);
    return 0;
}
char b [strlen (argv [1])];
char a [strlen (argv [1])];
int ij=0;
while (ij!=strlen (argv [1])) {
a [ij] =argv [1] [ij];
b [ij] =argv [1] [ij];
ij++;
}
a [ij] ='\0»;
b [ij] ='\0»;
revstring (a);
printf (»%s\n», a);

int i;
int j;
char c;

while (strcmp (a, b)!=0) {
i=1;

while (a [i]> a [i-1]) {
i++;

}

j=0;
while (a [j] <a [i]) {
j++;
}
```

```
c=a [j];
a [j] =a [i];

a [i] =c;

char x [strlen (a) +1];
char y [strlen (a) +1];
strcpy (x,a);
strcpy (y,a);

subb (x, i);

revstring (x);

subb2 (y, i);
sprintf (a, "%s%s», x,y);

printf (»%s\n», a);

}

}
```

Дополнение

Код исправлен, сравнение с i-1 больше не приводит к выходу за пределы массива. Также приятно сознавать, что данная реализация работает по времени сопоставимо (приемлемо, а может, и быстрее для малых n) с рекурсивным алгоритмом на том же Си. Сравнение проводилось с кодом по следующей ссылке:

https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/

Рекурсивный алгоритм выдал время работы для n=11:
real 2m9.213s
user 0m2.920s
sys 0m26.290s
Алгоритм из этой заметки выдал для n=11:
real 2m15.510s
user 0m19.750s
sys 0m34.300s

Для n=10
Рекурсивный:
real 0m11.919s
user 0m0.340s

sys 0m2.390s

Из этой заметки:
real 0m12.128s
user 0m1.490s
sys 0m3.040s

7) К рекурсии через перестановки

Поскольку речь пойдет о рекурсии, начну с конца со списка использованной литературы:

1) хорошая общая статья о рекурсии: habrahabr.ru/post/256351 (в ней автор говорит, что рекурсивный код легче для восприятия. Честно говоря, пока я не готов согласиться с таким выводом, именно поэтому появилась эта заметка).
2) разбор работы рекурсии на «самом низком уровне», тут много ассемблера, но всё достаточно понятно: club.shelek.ru/viewart.php? id=205 (особенно советую обратить внимание на тот момент, где идет речь об адресе возврата. Этот эпизод сильно облегчает понимание).

Лирическое отступление

Данная статья настолько рекурсивная, что написана автором для самого автора, а также для тех пользователей, которые, как и автор, не уверены в стопроцентном понимании данной темы.

А теперь приступим

Такой вот код генерации перестановок был найден мной на Stackovereflow (https://stackoverflow.com/questions/5506888/permutations-all-possible-sets-of-numbers). Памятуя о законе потери информации о том, что многие любят видеть всё в одной статье, перепечатаю код здесь (ссылка есть, алгоритм из учебника). На мой взгляд нижеприведенная конструкция имеет важную особенность — её очень легко понять и разобрать по кусочкам. Кроме того, скрипт можно значительно упростить, чтобы добраться до семечка — рекурсии. Начнём откусывать от кода.

Сам код

```
function permute ($str, $i, $n)
{
if ($i == $n) print «$str\n»;
else
{
for ($j = $i; $j <$n; $j++)
{
swap ($str, $i, $j);
permute ($str, $i +1, $n);
swap ($str, $i, $j); // backtrack.
}
}
}

// function to swap the char at pos $i and $j of $str.
```

```
function swap (&$str, $i, $j)
{
$temp = $str [$i];
$str [$i] = $str [$j];
$str [$j] = $temp;
}

$str = «0123»;
permute ($str, 0, strlen ($str)); // call the function.
```

```
Время выполнения исходного скрипта для n = 9:
    4.14418798685.
Исходный код выводит перестановки почти
    в лексикографическом порядке, а хотелось бы
    строго в нём.
Приступим к декомпозиции.
Откусим второй вызов функции обмена — swap.
 Смысл второго вызова в том, чтобы за один цикл
    сделать два обмена.
```

Но почему два счётчика, шеф?!

Количество циклов от этого не сокращается, только увеличивается количество операций.

Откусываем… и вдруг чудо! Вывод перестановок теперь в лексикографическом порядке.

А для одного вызова swap с тем же n = 9 время выполнения =2.76783800125.

Отмечу, что разница заметна даже для n = 8. Отлично! Чего бы с какой бы стороны еще откусить?

Откусим вызов функции и отправим операции обмена прямо в цикл.

```
function permute ($str,$i,$n) {
```

```
if ($i == $n)
print «$str <br/>»;
else {
for ($j = $i; $j <$n; $j++) {

$temp = $str [$i];
$str [$i] = $str [$j];

$str [$j] = $temp;

permute ($str, $i+1, $n);
}
}
}

$str = «123»;
permute ($str,0,strlen ($str));
```

Да как же ж так можно?! Да где же это видано, чтобы функции брали и откусывали?!

Если Вы когда-нибудь покупали подержанный автомобиль на рынке, то часто на свои замечания по поводу состояния машины могли слышать фразу: «На скорость не влияет!»

А вот и нет! Все-таки влияет. И пролить свет на это может статья по второй ссылке.

Результат времени выполнения нашего огрызка улучшился. 1.91801601648.

И код теперь совсем как на ладони.

Уберём единственную проверку из функции. Вывода станет заметно больше, (немножко припевов/повторов скрасит путь к рекурсии). При n = 9

С выводом в браузер уже возникают проблемы. И это всего

лишь при 986409 циклах. Здесь уместно вызвать функцию напоминания напомнить про ссылку на первую статью.

Но мы добрались до главного, до нашего семечка — рекурсии. Посмотрим, какие значения принимают переменные i и j. К этому мы и подбирались.

> — думаю, момент с изменениями значений переменных и есть основная трудность в понимании рекурсивного алгоритма перестановок. Уберём вывод и обмен, сократим n до 2.

Но как же понять, что происходит с переменными?!

Напечатаем их в цикле. Добавим в цикл для наглядности вывод i и j:

```
function permute ($str,$i,$n) {

for ($j = $i; $j <$n; $j++) {
echo «i=». $i.» j=». $j.»<br/>»;

permute ($str, $i+1, $n);
}

}

$str = «01»;
permute ($str,0,strlen ($str));
```

Получим вот такой вывод:

```
i=0 j=0
i=1 j=1
i=0 j=1
i=1 j=1
```

В котором всё сразу становится понятно. Так и хочется назвать это таблицей истинности.

Наш взгляд, привыкший к циклическим выводам, «запутывается в листьях и ветках».

Только веток у нас всего две, так что постараемся выбраться.

На самом деле все не просто, а очень просто: там где i =0 — это первая ветка, т.е. i =0 j=0 и i=0 j=1 — это первый вызов функции — наш ствол. Но поскольку вся программа рекурсивная, то при n = 2 вывод естественно через строчку.

А если n будет больше?

Тогда вначале мы увидим кусочек нашего ствола (i=0), а потом листья, листья, листья и где-то через n+x строчек снова мелькнет наш ствол. При выводе это может создать путаницу.

Заметим также в случае с перестановками, что поскольку в самом начале j принимает значение i, то на первых этапах выполнения программы видимой транспозиции элементов не происходит, хотя фактически обмен выполняется.

Какой же из всего этого вывод?

А вывод уже был в конце статьи, что по первой ссылке в начале этой заметки.

В итоге

Мы сумели осуществить несколько задач: сократить исходный скрипт, провести некоторые тесты скорости. Вернуть перестановкам истинный порядок. И, надеюсь, сумели разобраться с рекурсией. Можно было бы еще для наглядности нарисовать рекурсивное древо, но оставлю это на долю воображения читателя. Напоследок напомню про вторую

ссылку в начале заметки, для совсем бесповоротного понимания рекурсивных перестановок можно данный код использовать в качестве примера при работе с указанным материалом.

Весь алгоритм кратко в виде анимации (будем удерживать в голове тот факт, что в один момент времени выполняется один участок программы — один шаг. В качестве вспомогательной мнемоники можно представить себе, что «указатель процессора» в один момент времени находится в такой-то точке — участке программы. Поскольку при новом вызове функции всегда хранится адрес возврата, то можно проследить, как наш условный указатель доходит до первого «листа» — n=4, а потом возвращается на несколько шагов и счётчик цикла -j увеличивается):

Послесловие или небольшое добавление

Но что же наша нерекурсивная реализация на PHP?! После ряда существенных доработок алгоритм, близкий к алгоритму Нараяны Пандиты, выдал для n =9 время выполнения 1.76553100348

Надо сказать, что сама реализация стала довольно прозрачной:

Нерекурсивная реализация алгоритма перестановок на языке PHP

```
$b=«0123456»;
$a=strrev ($b);

while ($a!=$b) {
$i=1;

while ($a [$i]> $a [$i-1]) {
$i++;
}
$j=0;

while ($a [$j] <$a [$i]) {
```

```
$j++;
}

$c=$a [$j];
$a [$j] =$a [$i];

$a [$i] =$c;

$x=strrev (substr ($a, 0, $i));
$y=substr ($a, $i);

$a=$x.$y;
print '<br/>»;

}
```

К вопросу о методах упрощения кода, в приведенной выше реализации можно убрать лишние переменные.

Нерекурсивная реализация алгоритма перестановок на языке PHP

```
<?php
$b=«0123»;

$a=strrev ($b);

while ($a!=$b) {
$i=1;
while ($a [$i]> $a [$i-1]) {
$i++;
}
$j=0;

while ($a [$j] <$a [$i]) {
```

```
$j++;
}
$c=$a [$j];
$a [$j] =$a [$i];
$a [$i] =$c;
$a=strrev (substr ($a, 0, $i)).substr ($a, $i); print $a.
    "\n»;

}
?>
```

8) Перестановки без формул. (Код PHP)

Перелистывая вопросы и статьи в Интернете, я обратил внимание, что эта простая на первый взгляд тема составляет некую трудность при составлении алгоритма. Попробую максимально просто объяснить себе и вам алгоритм генерации перестановок, вернее, один из возможных. Многие статьи, описывающие тему перестановок, начинаются с формул или теории общей комбинаторики. Отступим от этого канонического принципа.

Есть задача: требуется напечатать все перестановки четырех чисел:

1, 2, 3, 4.

Решение — сначала на листке бумаги

1) Посчитаем последовательно перестановки для одного элемента, для —

1.

Запишем результат:

1

Он равен единице, один элемент переставлять некуда, но мы запомним

результат, пригодится.

2) Посчитаем перестановки для двух элементов — 1 и 2

Запишем результат:

12

21

У нас две перестановки. Все перестановки из дух элементов равны двум. Теперь нужно посчитать для трех элементов — 1, 2, 3. Для этого возьмем наше новое число — 3 и подставим его в каждую строку к перестановкам для двух элементов. Будем подставлять для каждой строки последовательно так, чтобы это число — 3 — побывало на каждой позиции, т. е.: в конце строки, между каждым элементом и в начале строки. Начнем с конца строки. Для первой строки получим результат (в виде квадратной диагональной матрицы):

123

132

312

Для второй строки результат:

213

231

321

Запишем результат.

3) Для четырех элементов — 1, 2, 3, 4 осуществим все то же самое, что и на шаге два. Возьмем значения, полученные ранее и подставим цифру 4 для каждой строки в конце, между каждым элементом и в начале строки.
Снова начнем с конца:

<1 2 3 4> <1 3 2 4> <3 1 2 4> <2 1 3 4> <2 3 1 4> <3 2 1 4>

<1 2 4 3> <1 3 4 2> <3 1 4 2> <2 1 4 3> <2 3 4 1> <3 2 4 1>

<1 4 2 3> <1 4 3 2> <3 4 1 2> <2 4 1 3> <2 4 3 1> <3 4 2 1>

<4 1 2 3> <4 1 3 2> <4 3 1 2> <4 2 1 3> <4 2 3 1> <4 3 2 1>

Получим 24 перестановки. Легко проверить — факториал числа 4 равен 24:

4! = 1*2*3*4=24

Обратим еще раз внимание: мы берем результаты, найденные на предыдущем шаге, берем число выше на единицу, подставляем это число в каждую строку — тянем с конца строки в начало. Когда это число на первой позиции, мы берем следующую строку и повторяем действия. Получаем простой алгоритм перестановок в нелексикографическом порядке, который можно быстро перевести на машинный язык. Результат, кстати, сильно напоминает код Грея для перестановок, а также алгоритм Джонсона-Троттера. Хотя я не вникал сильно, но если интересно, то почитать о нем можно, набрав в поисковике «Коды Грея для перестановок».

Об авторском праве

Кстати, лично мне протягивание цифры по строке пришло в голову после одной ассоциации из жизни, эта ассоциация связана с плетением лаптей или корзинок, когда каждое новое кольцо образуется протягиванием лыка или ивового прутика через каждую вертикальную дугу.

А теперь попробуем реализовать этот алгоритм на PHP, для хранения значений будем использовать файлы. Создадим два файла с именами 1.txt и 2.txt, в файл 1.txt запишем единицу.

Для каких-то практических задач использование нижеприведенного когда не планируется, поэтому навернем в нем всего побольше:

— Будем читать файл 1.txt построчно.

— Оборвем цикл, если строка пустая.

— Строку будем разбивать и хранить в массиве (explode).

— К ней добавим следующее число (array_push).

Неприятный факт, но еще на каждой итерации самого верхнего цикла будем использовать array_trim, так как в массиве у нас неизвестным образом появляется символ пробела.

— В цикле while будем использовать только list для смещения числа по строке.

— Все результаты будем писать в файл 2.txt, затем удалим 1.txt и переименуем 2.txt в 1.txt, обновим страницу, все предельно просто.

Основной упор в этой заметке не на код ниже строчки, которую вы сейчас дочитываете, а на тот алгоритм, который описан выше.

```
<?php

$handle = fopen («1.txt», «r»);

$handle2 = fopen («2.txt», «w+»);

while (!feof ($handle))

{

$ar = array ();

$line = fgets ($handle);

if ($line == «»)

{

break;

}

$ar = explode (».», $line);

$c = count ($ar);

array_push ($ar, $c +1);

$c+= 1;
```

```
    $ar = array_map ('trim', $ar);
    echo '<br />»;
    $s = implode (».», $ar);
    echo $s;
    fwrite ($handle2, «$s\r\n»);
    while ($c!= 1)
    {
    $c — ;
list ($ar [$c-1], $ar [$c]) = array (
$ar [$c],
$ar [$c-1]
);
echo '<br />»;
$s = implode (».», $ar);
echo $s;
fwrite ($handle2, «$s\r\n»);
}
}
```

```
fclose ($handle);

fclose ($handle2);

unlink («1.txt»);

rename («2.txt», «1.txt»);

?>
```

Post Scriptum, ссылки и немного истории

О нерекурсивном лексикографическом способе (в словарном порядке) генерации перестановок можно посмотреть статьи, раскрывающие смысл алгоритма индийского математика XIV века Нарайаны Пандиты. Видимо, он один из первых составителей нерекурсивного алгоритма.

Об истории комбинаторики в разных странах:
www.williamspublishing.com/PDF/978-5-8459-1158-2/part. pdf

Методы перестановок можно посмотреть здесь:
study.sfu-kras.ru/DATA/docs/Program...rs/gn_trans.htm
Для изучения вопроса о перестановках на php хотел бы отметить вот эту статью товарища tvolf, очень пригодилась (Спасибо огромное): tvolf.blogspot.ru/2013/09/php.html

Post Post Scriptum

Вдобавок перестановки можно генерировать с помощью псевдослучайных чисел — RANDOM, правда — этого долго, но все же такой способ есть.

— еще один из способов напечатать все перестановки для

числа n — сначала сгенерировать все размещения с повторением, а затем удалить значения, в которых символы повторяются — это самый простой для программирования, но самый долгий способ. Его обычно даже не рассматривают, но он все же есть, так как (напомню) перестановки — это частный случай размещений.

Выводы

В результате изучения комбинаторных алгоритмов и размышлений об их применении в современных информационных науках в гуманитарном ключе автор пришёл к выводу, что подобные алгоритмы могут быть использованы:

1) в библиотечных системах, для комбинаторного поиска сочетаний слов в тексте или релевантного поискового вывода.

2) как вспомогательные гипотетические средства при попытках дешифровки древних (пока неизвестных) алфавитных, слоговых или
созданных клинописью текстов. Эта теория не проверялась на практике, но данный метод явно выводится умозрительно при поверхностном изучении переборной комбинаторики применительно к тексту.

3) для иной работы с языковыми единицами, например, для целей
этимологии — поиска слов (возможно) с одним историческим корнем в одном или даже разных языках.

4) а также для иных целей: порождения и поиска разных сочетаний слов, цветовых сочетаний (схем) и т. д.

5) Возможно, для порождения искусственных языковых единиц.

*** В книге используются языки программирования C, C89, PHP.**

Дополнение

Ниже алгоритм порождения всех перестановок без повторений итеративным способом на языке C89, сокращённый максимально для простоты понимания.

* Заметим, что для внешнего цикла используется декремент f, что напоминает способ организации циклов на языке Ассемблера:

```
#include <stdio. h> </stdio. h>

int main () {
char a [] = «4321»; //array

int i, j;

int f=24;

//factorial

char c;

//buffer

while (f — ) {
printf (»%s\n», a);

i=1;

while (a [i]> a [i-1]) i++;

j=0;
```

```
while (a [j] <a [i]) j++;

c=a [j];

a [j] =a [i];

a [i] =c;

i — ;

for (j = 0; j <i; i — , j++) {
c = a [i];

a [i] = a [j];

a [j] = c;}}
}
```

Оглавление

7) К рекурсии через перестановки.
8) Перестановки без формул. (Код PHP).
9) Выводы.

Иван Гаврюшин

Гуманитарные основы комбинаторных алгоритмов

Создано в интеллектуальной издательской системе Ridero