Make it

Theory workshop

Generating simple permutations

Permutations are easy to describe, after all they're just anagrams, but how do you generate them?

In this issue...

▶ **WHAT'S COVERED**

Writing a program that generates all the permutations of a given set of symbols, like ABCDE, seems like it should be easy. After all defining a permutation is simple enough. But, in reality, writing such an implementation is more difficult than you think, and then you have to worry about whether your code actually generates the complete list of permutations and doesn't duplicate any.

One of the hardest things to implement, it seems, is creating all the permutations of a series of symbols. I remember, back when I was learning about algorithms, someone wrote to me asking how it was done, causing me to spend far too long re-inventing the wheel; a problem that nowadays is much reduced thanks to the newer technologies of the Internet and Google.

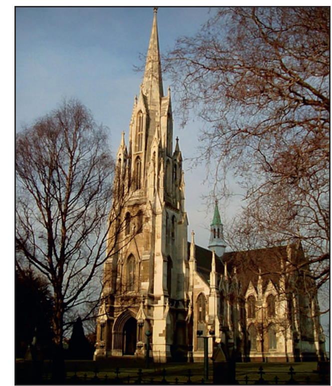
To begin, let's review what a permutation is (as compared to a combination, say). A permutation is a rearrangement of a set of symbols or objects. To give an example, if we take the symbols A, B and C, then one permutation is ABC, another is CBA and so on.

Since there are a possible three symbols for the first position, and having fixed that, and two for the second position, and having fixed that, only one for the third position, means that there are a total of six possible permutations available for our three symbols (3 x 2 x 1). Using our example, these are ABC, BAC, BCA, CBA, CAB and ACB. Following the same logic, there are 24 permutations of four symbols, 120 permutations of five, and so on. In general, I'm sure you can see that, for N symbols, there are N! (read, N factorial) different permutations (where N! = N x N-1 x N-2 x ... x 2 x 1).

Now, the biggest issue we should realise when generating permutations from here is that the factorial function grows incredibly quickly as N begins to grow. The table in Figure 1 shows the growth of N! for N between 10 and 20. The table also shows a couple of columns against each value of N: – the time it takes to generate that number of permutations. Here, we're assuming a computer can generate and use a million permutations every second. This could be easily double with an average PC of today's ability.

Recursive backtracking

Moving up to more high-end equipment, most can generate a thousand-million permutations every second. There's also a column in Figure 1 which shows pipe-dream level performance and what could be achieved. I've colour coded the chart to go from green to red to indicate the speed. Green means you'll get an immediate answer. Yellow means you'll have to get a coffee or possibly go on holiday as you wait for it to finally finish. Red means your computer will wear out before it's complete.



▲ **The old English tradition of bell ringing has brought about an algorithm we can learn from.**

Viewing the chart, I'm sure you can see that talking about permutations of 12 to 14 symbols is about as many as we can hope to discuss here. I would advise you that, as you experiment with the following algorithms, you stick to four, or maybe five, symbols. Using such a small number will give you a feeling for the algorithm and also mean that

N	Number of Permutations	Million/sec	Billion/sec	Trillion/sec
10	3628800	4 sec	instant	instant
11	39916800	40 sec	instant	instant
12	479001600	8 min	0.5 sec	instant
13	6227020800	2 hrs	6 sec	instant
14	87178291200	1 day	1.5 min	instant
15	1307674368000	16 days	22 min	1 sec
16	20922789888000	8 mths	6 hrs	21 sec
17	355687428096000	11 yrs	4 days	6 min
18	6402373705728000	too long	2.5 mths	2 hrs
19	121645100408832000	too long	4 yrs	1.5 days
20	2432902008176640000	too long	too long	28 days

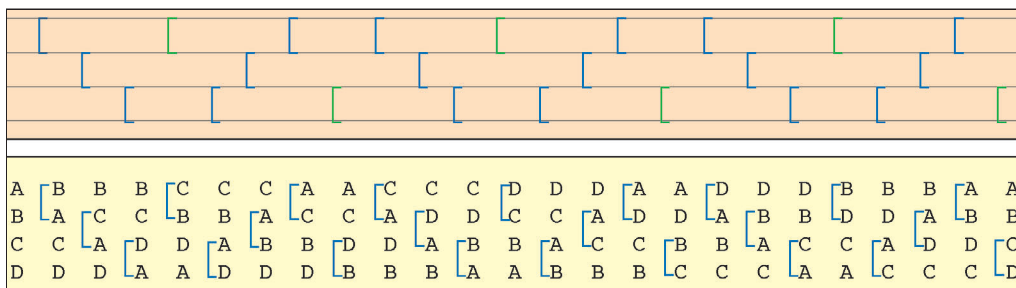
▲ **Figure 1: As the number of symbols increases, the factorials grow really quickly.**

Nugget

In the old days, strings were mutable, but in modern run-time libraries like .NET they are not. This means that you can't implement permutation generators using a string to hold the symbols, since you will find that you won't be able to swap over the characters in the strings. In these kinds of run-times, you will have to hold the symbols in an array of characters and may have to write a method to print it.

Other run-times have mutable strings; for example, Delphi has mutable long strings and short strings. With these compilers and libraries you will be able to implement the algorithms in a more familiar medium.

PCP263.theory 146 1/11/07 16:50:46



▲ **Figure 2: The permutations used by bell ringers when performing plain changes.**

your implementation won't take that long to complete.

The first algorithm for generating permutations that people encounter is a recursive backtracking algorithm, even though they may not name it as such. This is the one I devised to answer the question I was asked.

We assume that there is an array of symbols that we have to generate all the permutations of. It's given to us in the initial order (which is obviously one of the permutations). We also assume two routines, one called 'Swap' which swaps over two elements of the array, given their indexes; and one called 'Process' which does something with the array, such as print it to the screen.

The backtracking algorithm works by exchanging each element in the array with the end one, and then recursively calling itself on the remainder of the array. Once the recursive call ends, the two elements are swapped over again. So, in pseudo code, the 'Generate' routine looks like this:

- Pass in the number of elements to permute, call this N
- If N is 1, call Process on the whole array, then exit
- Otherwise, enter a for loop with a counter C = 1 to N and:

```
- Swap(C, N)
- Generate(N-1)
- Swap(C, N)
```

The one thing that can be said for this routine is that it doesn't use any extra memory to hold data, apart from the minor loop variable and so on. Unfortunately, it also requires 2 x N! calls to the 'Swap' routine.

Plain changes

Incredibly, though, that's not the first algorithm devised to solve this particular problem. Way back in the 1600s in England, church bell ringers became interested in what was known as change ringing. This is the art of ringing bells of different notes in pleasing sequences, with the sequences being known as changes. No attempt was made here to play a tune, like in campanology, but instead to merely ring the bells in various permutations.

The main reason for ringing the changes rather than playing a simple melody is that the bells concerned are heavy and require a separate ringer for each. Also, because they have a lot of inertia once set in motion, the changes that were devised needed to allow the ringer time to prepare for their next ring. The first book on such change ringing was published back in 1668.

The simplest algorithm that bell ringers devised was the one known as plain changes. After each round, two ringers swap places in the ringing sequence. They don't swap physical places, just the order in which they play in the next round. A permutation of ringers, in other words.

Essentially, in plain changes, the first bell moves down the sequence of bells, being inserted in between each other bell. Once it reaches the end, it then sweeps back up the sequence of bells. Every time it reaches the end, the two bells at the opposite end of the sequence change places, before the first bell moves back in the opposite direction.

A picture would help explain this further, so in the bottom part of Figure 2 (above), I show the whole set of permutations of four symbols, with indications of which two elements are swapped in each cycle. So in the first four changes, A swaps over with its nearest neighbour and makes its way down to the final position. At that point the two elements at the beginning of the list are swapped over. After this, A makes its way up to the top again. Once it reaches there, the two elements at the opposite end are swapped, and then A makes its way down again. Incredibly, after six back and forths like this, you get the initial permutation again, having worked your way through each of the others.

The top part of Figure 2 illustrates how close these changes can look like music, but also illustrates more effectively the changes in direction that symbol A makes as it moves up and down.

This algorithm actually does a lot better than the first backtracking algorithm, since it only requires N calls to the 'Swap' routine, half as many as before. Implementing the algorithm, known as the 'Johnson-Trotter Algorithm', in an efficient manner is however decidedly tricky. I won't implement it here (it would exceed the limits of this article), but instead I sketch out how it works in the 'Spotlight on' section.

I hope this has shown that, despite the ease with which you can define and understand permutations, how difficult it is to devise an efficient algorithm to iterate all of them of a certain size. There are many other ways of doing this kind or work, but the winner in terms of speed, at least at the time of writing this, is one called 'Heap's Algorithm', where Heap is the computer science researcher and not the same heap as in heapsort. ■

Julian M Bucknall has worked for many major companies and is now CTO for Developer Express.
feedback@pcplus.co.uk

Nugget

A fun little program to write is one which generates the anagrams of another word. To do this you'll have to have a word list (there are several available online) to use for checking your anagrams. At its simplest, the game plan works a little like this: generate all the permutations of the letters in the given word, and search the word list for each permutation. Rather than sequentially search the word list, you should first throw away any words over 10 letters, say, and then insert the remainder in a hash table. Checking anagrams then becomes a matter of verifying that the current permutation is present in the hash table. ■

Spotlight on... Johnson-Trotter

A simple way to generate permutations of length N is to start with all the possible permutations of length N-1, and insert the next symbol in each gap in each of the smaller permutations, as our example below shows:

```
Nabc...z
aNbc...z
abNc...z
...
abc...Nz
abc...zN
```

The symbol N seems to travel rightwards along the sequence of previous symbols as you look down the list. If the permutations were listed in the reverse order, the symbol N would seem to travel to the left instead.

The 'Johnson-Trotter Algorithm' makes use of this 'travelling' metaphor. It assigns all symbols a 'direction' value, either left or right. It then makes a definition: a symbol is 'mobile' if it's larger than the immediate item it's pointing at. As you'll see the mobile symbols will swap places with symbols less than they are. Here is the Johnson-Trotter Algorithm: Initialise all symbols in lexical order (ABC...) pointing to the left. While there is at least one mobile symbol:

- Find the largest mobile symbol
- Swap it and the immediate symbol it is looking at
- Process the permutation
- Reverse the direction of all symbols larger than it. ■

PCP263.theory 147 1/11/07 16:50:55