Jan Keromnes - 严奇龙

An exercise in n! and backtracking for the generation of permutations

The problem

Our goal is to implement a **permutation generator** using the `C` language.

The input elements are given as parameters to the program, e.g.

```
1 2 3 4 5 6 7 8 9
```

or

```
a b c d e f.
```

The resulting permutations should be printed on `stdout`, making it possible to view them in the console or to redirect them to a text file.

The algorithm should be based on either switching method or inversion method, and further customization is encouraged.

For the purpose of simplicity, we are going to limit the arity of permutation elements to **64**. This is reasonable enough, because an input sequence of 64 elements would result in `64! = 1.2688693 × 10^89` permutations. Assuming that a 64 element output sequence takes up 64 bytes, the resulting text file would be `8.12076366 × 10^90 bytes = 6.71733826 × 10^66` yottabytes large. For comparison, the size of Internet was estimated to be half of one thousands of a yottabyte in 2009 (source: Wikipedia).

Intuition

As our program has to output n! permutations, we would like to establish a looping system that has n! iterations. As it would be too costly to implement a counter towards n!, we could split this counter into its factors.

This can be done, with backtracking in mind, as an n-sized array that counts in factors.

Examples:

```
0 → 0
1 → 1 = 1!

0 0 → 0
0 1 → 1
1 0 → 2 = 2!
```

```
0 0 0 → 0
0 0 1 → 1
0 0 2 → 2
0 1 0 → 3
0 1 1 → 4
0 1 2 → 5
1 0 0 → 6 = 3!
```

We will use an array of size n composed of zeros, and we will use it to count towards n! by starting with the right-most index and incrementing it until it reaches the value of its index (`array[4]` will not go past 4). Once it reaches its maximum value, we set it to zero again and increment the value at the previous index, and so on.

The array `[a b c d]` represents the number `a*2*3*4 + b*3*4 + c*4 + d`, therefore when we reach `[1 0 0 0]`, we know that we will have reached `1*2*3*4 = 4!`.

This array will be used to implement a backtracking algorithm to generate permutations.

Permutation technique

To generate all permutations, we will use the switching pattern, which is obtained by swapping two items at each iteration:

```
1 2 3
1 3 2
3 1 2
3 2 1
2 3 1
2 1 3
```

Its goal is to make the nth element move all the way trough permutations of n-1 elements in order to generate the permutations of n elements.

We are going to use the previously introduced backtracking array to generate the swapping pattern. While counting up toward 3! with our array, incrementing the 2nd index results in swapping the 2, and incrementing the 3rd index results in swapping the 3, etc.

Example:

```
(array  → permutation)

0 0 0 0 → 1 2 3 4 (given input sequence)
0 0 0 1 → 1 2 4 3 (incrementing 4th index = swapping the 4)
0 0 0 2 → 1 4 2 3 (again)
0 0 0 3 → 4 1 2 3 (again)
0 0 1 0 → 4 1 3 2 (incrementing 3rd index = swapping the 3)
0 0 1 1 → 1 4 3 2 (back to incrementing 4th index)
0 0 1 2 → 1 3 4 2 ...
0 0 1 3 → 1 3 2 4 ...
0 0 2 0 → 3 1 2 4
0 0 2 1 → 3 1 4 2
0 0 2 2 → 3 4 1 2
0 0 2 3 → 4 3 1 2 ...
```

```
0 1 0 0 → 4 3 2 1 (incrementing 2nd index = swapping the 2)
0 1 0 1 → 3 4 2 1
0 1 0 2 → 3 2 4 1
0 1 0 3 → 3 2 1 4
etc...
```

We define:

- `n` the number of elements
- `stack` the backtracking array of size `n`
- `esp` the stack pointer (index we are currently incrementing)

Factorial oddity

From the previous sections, we know how to determine which element from the input sequence we need to move. However, we are still missing information to determine exactly which indexes we should swap:

- We need to know in which **direction** an element is moving (right to left, or left to right).
- Sometimes when we are moving smaller elements, previously moved bigger elements might be in our way, causing an **offset**.

Direction

The direction of an element is directly related to the oddity of times it has travelled from one end to another. Since we start on the right of the array, if an element has travelled an **even** number of times from end to end, then it is still at the right end and will move to the left. If it has previously travelled an **odd** number of times, then it is at the left and need to go to the right.

The number of times an element n has travelled before can be determined by the number of times its previous element n-1 has been incremented. This in turn can be calculated as `stack[n-1] + n*stack[n-2] + n*(n-1)*stack[n-3] + n*(n-1)*(n-2)*stack[n-4] + ...` (see the last example in *Intuition*).

Since we are only interested in the oddity of this number, we can discard all terms that are multiplied by `n*(n-1)`, because they will always be even. This leaves us with having to determine the oddity of the following expression:

```
stack[n-1] + n * stack[n-2]
```

If it is *even* we move from right to left; if it is *odd* we move from left to right.

Offset

When we are moving an element, we know its index relative to the left-most element. However, if bigger elements were moved all the way to the left, we have to use an offset. The offset equals the number of bigger elements that were moved to the left.

To determine the number of bigger elements to the left of our element, we need to look at the number of times they travelled from right to left and back. Again, if this number is *odd*, the element is at our left causing an offset, but if it is *even* it is at our right and we can forget about it.

The number of times the immediately bigger element has travelled is determined by the oddity of our current number. As explained before, this number is:

```
stack[n] + (n+1)*stack[n-1]
```

and if it is odd we set our offset to `1`.

If the first bigger element is on the left, the second bigger element can also be there. This can be determined by looking at the oddity of the following expression:

```
n+2
```

If it is odd, we add `1` to our offset. Subsequent bigger elements are discarded because they will have travelled `(n+2)*(n+3)*x` times, which is always even.

Example:

Say we are working on `n = 4` numbers. If we are currently swapping the number 2, and if the number 3 has travelled 1 time, 3 is on the left and causes an offset of 1. The number 4 has then travelled `3*1` times, which is also odd and it is therefore also on the left. The offset is then 2.

```
1 2 3 4 (moving the 4)
1 2 4 3 ..
1 4 2 3 ..
4 1 2 3 ..
4 1 3 2 (moving the 3)
1 4 3 2 (moving the 4)
1 3 4 2 ..
1 3 2 4 ..
3 1 2 4 (moving the 3)
3 1 4 2 (moving the 4)
3 4 1 2 ..
4 3 1 2 ..
```

At this point, we should move the `2`.
`3` has travelled once from right to left, and is to the left of `1 2`.
`4` has travelled `3*1` times from right to left, and is also to the left of `1 2`.
Therefore, our offset is `2`.
As we know that `2` has moved `0` times, which is even, it should move from right to left.
We should swap the indexes `1` and `0`, with an offset of `2`, thus `3` and `2`:

```
4 3 2 1
```

If there are no bigger elements, they are obviously discarded from the offset:

```
1 2 3
1 3 2
3 1 2
```

Here, only `3` is in front of `1 2`. There is no element `4`.

Therefore, the offset is only `1`, and so we swap indexes `2` and `1`:

```
3 2 1
```

General formula for indexes to swap

Finally, there are two formulas to determine which index to swap.

If we are moving from right to left:

```
swap ( offset + esp - stack[esp], offset + esp - stack[esp] + 1 )
```

If we are moving from left to right:

```
swap ( offset + stack[esp] + 1, offset + stack[esp] + 2 )
```

This step completes our implementation of the permutation generator.

Usage

The implemented program is called `brute`.

It can be compiled using the command:

```
make
```

And it can tested with the command

```
make test
```

To use it, simply call it with a sequence of elements to permutate as arguments:

```
./brute 1 2 3 4 5 6 7 8 9
./brute A C G T
```

To redirect permutations to a file:

```
./brute 1 2 3 4 5 6 7 8 9 > perm9.txt
```

To mute the output:

```
./brute 1 2 3 4 5 6 7 8 9 > /dev/null
```

Performance

To measure the performance of our program, we use the unix command `time` as follows:

```
time ./brute 1 2 3 4 5 > /dev/null
```

Here are the results for different input sizes:

```
    -- 6 elements --
    real  0m0.004s
    user  0m0.000s
    sys   0m0.000s

    -- 7 elements --
    real  0m0.014s
    user  0m0.012s
    sys   0m0.000s

    -- 8 elements --
    real  0m0.044s
    user  0m0.040s
    sys   0m0.000s

    -- 9 elements --
    real  0m0.382s
    user  0m0.376s
    sys   0m0.008s

    -- 10 elements --
    real  0m4.120s
    user  0m4.112s
    sys   0m0.000s

    -- 11 elements --
    real  0m50.040s
    user  0m49.923s
    sys   0m0.064s

    -- 12 elements --
    real  11m2.327s
    user  11m0.901s
    sys   0m0.720s
```

As we can see, the program is performing quite well. But we should try to make it faster.

Parallelizing

We define `stop` the index at which to stop generating permutations. For `stop = 0`, the algorithm is the same as previously shown. For `stop = 1`, the algorithm swaps all the elements but stops before swapping `1` and `2`. This can be used to split the task between multiple threads.

If we define `stop = 1`, and we spawn one thread to work on `1 2 3 4 5...` and another to work on `2 1 3 4 5...`, then in the end we will have generated all the permutations using two threads.

This idea can be generalized: We generate all the permutations of elements from 0 to `stop`, disregarding the rest of the elements, and give the generated solutions to different threads to permutate the rest of the elements.

This technique has been implemented in `brute`, with support for up to 2 threads. Defining `FORK = 1` will set `stop = 1` and spawn a secondary thread. As we can see, the gain in performance is welcome:

```
    -- 10 elements -- (previously ~4s)
```

```
real  0m2.400s
user  0m2.392s
sys   0m0.004s

-- 11 elements -- (previously ~50s)
real  0m29.269s
user  0m29.178s
sys   0m0.040s

-- 12 elements -- (previously ~11m)
real  6m25.722s
user  6m24.576s
sys   0m0.452s
```

Conclusion

We successfully implemented an efficient permutation generator in `C`, and we managed to make it faster by running on multiple threads (scalability is the modern way of making programs run faster).

On a final note, we could think of different ways to make it even faster. One idea would be to use a boolean array to represent the oddity of each factor, instead of using mathematical formulas to determine them at every iteration. This array would be updated after each iteration by switching boolean values "on" and "off", replacing several CPU cycles by an array access, as could add up significantly less when the number of iterations n! becomes bigger. Additionally, as we chose to limit our input size to 64 elements, we could even represent this boolean array as a 64 bit integer, on which we could then perform bitwise operations to memorize the oddity of each factor.