Perl Golf

Dave Hoover, dave@redsquirreldesign.com

Abstract

Prior Perl Golf columns focused exclusively on winning solutions. A number of golfers requested coverage of a more down-to-earth solution. This month's deconstruction covers a solution near the middle of the field. May's challenge featured the Kolakoski sequence, a series of self-describing numbers.

1 The Challenge

The Kolakoski sequence is an infinite series of alternating self-describing numbers.

```
% perl kola.pl 2 3 20
22332233223322332233
```

The string's first digit (2) determines the length of the first block (22). The second digit (2) determines the length of the second block (33). The third digit determines the length of the third block. Blocks alternate between the two different digits. This continues indefinately. Solutions need only handle at least 500 characters in the sequence, however

Programs take three numbers from the command line—two digits which determine the construction of the sequence (e.g. 2 & 3) and the length of the output string (e.g. 20).

2 The Deconstruction

Philippe 'BooK' Bruhat finished with a 75 stroke Kolokoski solution which put him slightly ahead of the median 79 stroke solution. He submitted 25 working versions over 6 days, which gives us an excellent history on the development of his final solution:

```
#!perl -l
$@.=($.=$ARGV[$_%2])x(substr$@,$_,1 or$.)for 0..500;
print substr$@,0,pop
```

When golfers test several algorithms, they better understand the options available. Novice golfers frequently trap themselves in their own cleverness hindering them when they quickly discover a working solution and become emotionally attached to the algorithm. A distinguishing factor between good and great golfers is the ability to throw away an algorithm after golfing it to its limit.

We reformatted the final solution to make it easier to read. The special variables $@ (eval error) and $. (input line number) become $string and $first, respectively. Next, we reformat the foreach loop by breaking it down into four explicit statements with comments, and add clarifying whitespace.

```
#!perl -l
$first = $ARGV[0];                      # Copy the first argument.
foreach $i (0..500) {                   # 501 iterations.
    $digit = $ARGV[$i % 2];             # Alternate digits.
    $length = (substr $string, $i, 1    # Get block length from string.
                  or $first);           # If no string, use first arg.
    $string .= $digit x $length;        # Build the output string.
}
print substr $string, 0, pop;           # Output the correct length.
```

With this reformatting, the underlying algorithm becomes more recognizable. As Perl Golf progresses, veteran golfers recognize the importance of compressible algorithms and place as much value on them as they do on other techniques.

The shebang line contains the -l switch which causes all print statements to automatically append a newline. XXX: reference to earlier column.

The loop has to determine two things to output the next part of the sequence—the digit to use and the length. Philippe determines the length of the next block by reading from the specified location in the string he simultaneously builds. On the first iteration, the string is not yet defined, so substr returns false. The short circuit or operator returns the value of $first which he assigns to $length. On subsequent iterations, the second digit in $string is the length of the second block, the third digit the length of the third block, and so on until the loop finishes.

$i % 2 is a simple way to test whether a number is odd or even. It returns 1 if $i is odd and 0 if even. With each iteration, $i will switch from odd to even, so Phillipe uses it to alternate between $ARGV[0] and $ARGV[1]. This value is the next digit to use.

Once Phillipe has the next digit and the length, he can add to the string. The repetition operator, x, repeats its left operand (the next digit) by the number its right operand (the length) specifies. He adds this to $string, and loops again.

Philippe admits that he is emotionally attached to the substr function. In this solution, both instances of substr take three arguments—the first argument is the string to operate on, the second is the starting point within the string, and the third is the length of the substring to extract. He uses substr to output the string with the correct length (from $ARGV[2]), which pop returns. A pop without arguments in the main body of the program removes the last element from @ARGV (though in subroutines it operates on @_) and returns it—in this case using it as the third argument to substr. With the same input as the fist example, Phillipe constructs a string 1250 characters long, although he only needs the first 20. The substr function extracts the first 20 characters and he never uses the rest of the string, affirming that brevity, not efficiency, was his primary concern.

With the mechanics explained, we begin to return the solution to its original form by removing unnecessary white space, shortening variable names, and replacing foreach with its shorter synonym, for. We shorten the variable names $first, $digit, $length, and $string to only their first letters, $f, $d, $l, and $s.

```
#!perl -l
$f=$ARGV[0];
for$i(0..500){
$d=$ARGV[$i%2];
$l=(substr$s,$i,1 or$f);
$s.=$d x$l;
}
print substr$s,0,pop;
```

Next, we replace the $d and $l of the string-building statement `$a.=$d x$l` with the expressions that define them. This consolidation reduces the for loop's block to one line. We discard $i since we can use the default index variable $_.

```
#!perl -l
$f=$ARGV[0];
for(0..500){
$s.=$ARGV[$_%2]x(substr$s,$_,1 or$f);
}
print substr$s,0,pop;
```

Since the for loop has a single statetemt, we can rewrite the loop as a statement modifier.

```
#!perl -l
$f=$ARGV[0];
$s.=$ARGV[$_%2]x(substr$s,$_,1 or$f)for 0..500;
print substr$s,0,pop
```

Explicitly using @ARGV two times should tip off a golfer that he can still make improvements. On the first pass, $ARGV[$_%2] is equivalent to $ARGV[0], meaning we can use it to define $f. The parentheses around `$f=$ARGV[$_%2]` force x to evaluate the result of that statement. The result of an assignment is the value assigned,

```
#!perl -l
$s.=($f=$ARGV[$_%2])x(substr$s,$_,1 or$f)for 0..500;
print substr$s,0,pop
```

Finally, we replace the reader-friendly `$s` and `$f` with Philippe's original `$@` and `$.`.

```
#!perl -l
$@.=($.=$ARGV[$_%2])x(substr$@,$_,1 or$.)for 0..500;
print substr$@,0,pop
```

3 References

Kolakoski sequence – http://mathworld.wolfram.com/KolakoskiSequence.html

Perl Golf web site – http://perlgolf.sourceforge.net/

Kolakoski Perl Golf – http://perlgolf.sourceforge.net/TPR/0/3/