Working with Bit Vectors

brian d foy
brian.d.foy@gmail.com

Perl is a high level language, so I don't have to play with bits and bytes to get my job done. The trade-off, however, is that I have to let Perl manage how it stores everything. What if I want to control that?

Memory Bigfoot

Perl trades memory for speed, meaning that it will gladly gobble up memory, waste it even some might say, so it can run faster. By keeping more stuff in memory, it can easily look them up. Most of the time that's a good thing. I can easily create arrays and hashes, manipulate them, and so on. I don't have to think about it.

Perl containers (or aggregates, which are terms for things that can have more than one item) group scalars. Scalars can be of any size, up to available memory, but no matter their size, they have a bit of overhead. Even a scalar variable with no value takes up some space. The Devel::Size module can tell me how much, although the actual numbers may vary on your platform and version of Perl (mostly since the size of various low-level data types, such as an int, depend on the architecture).

```
use Devel::Size qw(size);

my $scalar;

print “Size of scalar is “ .
  size( $scalar ) . “ bytes\n”;
```

On my Powerbook G4 running Perl5.8.4, this scalar takes up 12 bytes, and it doesn't even have a value.

```
Size of scalar is 12 bytes.
```

I could use Devel::Peek to see some of this.

```
use Devel::Peek;

my $scalar;

print Dump( $scalar );
```

The output shows me the Perl has already set up some infrastructure to handle the scalar value.

```
SV = NULL(0x0) at 0x1807058
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY)
```

Even with nothing in it, the scalar variable has a reference count and the scalar flags. Now, imagine an array of several hundred or thousand scalar values, each with their own scalar overhead.

Doing It Myself

I don't need to use Perl's arrays (or scalars) to store my data. If I have enough data, and another way to store it and then access it so I save a lot of memory and don't impact the runtime too much, I should use that. Perl more than one way to do this, of course. I'm going to spend most of my article showing bit vectors, but along the way I'll show a couple of related techniques.

First, the easiest thing I can do involves a really long string where each character (or other number of characters) represents an element. I'll pretend that I'm working with DNA (the biological sort, although you should probably use BioPerl for this sort of thing), and I'll use the letters T, A, C, and G to represent the base pairs that make up the DNA. Instead of storing the sequence as an array of scalar variable each holding one character (or even object representing that base), I store them as sequential characters in a string.

```
my $strand = 'TGACTTTAGCATGACAGATACAGGTACA';
```

I can then access the string with substr(), which I give a starting position and a length.

```
my $codon = substr( $strand, 3, 3 );
```

I can even change values since I can use substr() as an lvalue.

```
substr( $strand, 2, 3 ) = 'GAC';
```

Of course, I can store hide these operations behind functions, or I can even make an object out of the string and call methods on it. That's subject for a different article, though.

One step up the sophisication ladder is pack(), which does much of the same thing but with much more flexilibity. I can shove several different types into a string and pull them out again. I'll skip the example and refer you to the Tie::Array::PackedC module which stores a series of integers (or doubles) as a packed string instead of their numerical and possibly string values in separate scalar variables.

A bit vector does the same thing as the single string or the packed string. In one scalar value, it stores several values. Just like in my DNA example, or the stuff that pack() does, it's up to me how I partition that bit vector and then represent the values.

Playing With Bits

Before I get started with the bit vector though, let's talk about bytes and bits. Most programmers already know that most computers are digital and store a sequence of 0's and 1's. Each of those is a bit. Eight bits make up a byte, and, extending the homophone a bit to far, half a byte is a nybble. In Perl, I can represent these literally (Perl 5.6 and later) using the 0b notation just like I can represent hexadecimal values with a leading 0x. I say a bit is turned on if it's 1, and turned off if it's zero.

```
my $all_bits_on  = 0b11111111; # same as 255
my $all_bits_off = 0b00000000; # same as 0
```

In Perl, I can insert underscores into numbers to make them easier to read. Perl simply ignores them. Instead of looking at a long line of bits, I break them up into manageable groups to make the positions easier to find.

```
my $all_bits_on  = 0b1111_1111; # still 255
my $all_bits_off = 0b0000_0000; # still 0
```

Going the other way, I can use sprintf() and printf() to print integers in binary format using the %b format specifier.

```
printf "The bit pattern is %b\n", 37;
```

This formats the argument, 37, into the sequence of bits that represent it.

```
The bit pattern is 100101
```

Perl also has several bit operators to do things to bits. They're the same as in C (and I usually pull out my favorite C book when I want to play with them) and documented in perlop's “Bitwise String Operator” section. Be careful in Perl though, since these same operators work differently on characters and numbers. Ensure you're using numbers!

Bitmasks

Once I have a sequence of bits, I need to select the bits that represent an element. I use a mask to hide everything that I don't want to see and show only the bits I want to see. When I bitwise & two values, my result only has the bits turned on in the positions they were both on in the two values. Ever wonder what the difference between “Perl” and “perl” is? According to the perlfaq, it's one bit, literally. When I & the first letters, I get back “P” again.

```
  1010 0000   # capital P
& 1110 0000   # little P
-------------
  1010 0000   # capital P again!
```

If I want to find out if I have the lowercase version (without doing it the easy way), I can use a mask to see if that extra bit is set.

```
  1?10 0000   # either P or p
& 0100 0000   # the mask
-------------
  0?00 0000   # true if p, false if P
```

Sometimes, I need to see all complement of a bit pattern. If I want to know all the bits that are unset, that's the same thing as toggling every bit and seeing which ones are then set. The bitwise negation operator, ~, does that for me. Watch out when you do this, since it does it on the full integer (which is 32 bits on my machine, meaning that all those high bits that weren't set become set). I fix that with a mask to select just the low eight bits.

```
my $neg = (~0b10101100) & 0xFF; # 0b01010011
```

The vec function

The builtin vec function treats a string as a bit vector made up of elements with the number of bits we specify, although that number has to be a power of two. It works like substr(), although it only works with one “element” at a time. Since vec() takes care of the storage for me, I let it figure out where to put everything, and that turns out to be a lot easier than doing it all at once since the behind-the-scenes storage gets pretty tricky when Perl starts converting endianness and other insanity. As long as I store them in the same way I access them (that is, with the same bit length), everything works just fine.

In my DNA example, I had four things to store (T, A, C, G). Instead of using a whole character (8 bits to store each one of those, I can use just two bits. In this example, I turn a 12 character string representing into a bit vector that is three bytes long.

```
my %bit_codes = (
   T => 0b00, 0b00 => 'T',
   A => 0b11, 0b11 => 'A',
   C => 0b10, 0b10 => 'C',
   G => 0b01, 0b01 => 'G',
   );

use constant WIDTH => 2;

my $bits = '';
my @bases = split //, 'CCGGAGAGATTA';

foreach my $i ( 0 .. $#bases ) {
   vec( $bits, $i, WIDTH ) =
      $bit_codes{ $bases[$i] };
   }

print "Length of string is " .
   length( $bits ) . "\n";
```

That's my bit vector of 12 elements, and now I want to pull out the third element. I give vec() three arguments: the bit vector, the number of the element (zero-based, like most other things in Perl), and the width in bits of each element. I use the value that vec returns to look up the base symbol in the hash (which maps both ways).

```
my $base = vec $bits, 2, WIDTH;
printf “The third element is %s\n”,
   $bit_codes{ $base };
```

Keeping Track of Things

Now that I know how to deal with bits and use vec, I can get to the point of my article. In Eric Maki's article in this same issue “Generating Sudoku”, he uses bit vectors to represent possible solution states. He represent each puzzle row with 9 bits, one for each square, and turns on a bit when that square has a value. A row might look like:

```
0 0 0 1 0 1 1 0 0 # pivot row
```

Once that square has a value, he eliminates all other candidates that also have values in that square. He knows which ones those are because a bitwise & of the two row returns a true value, just like my earlier “P” example.

```
  0 0 1 0 0 0 1 0 0 # candidate row
& 0 0 0 1 0 1 1 0 0 # pivot row
--------------------
  0 0 0 0 0 0 1 0 0 # true value, eliminate
row

  0 1 0 0 1 0 0 0 1 # still a candidate
& 0 0 0 1 0 1 1 0 0 # pivot row
--------------------
  0 0 0 0 0 0 0 0 0 # false, still in the
running
```

He also keeps track of all the rows he's no longer considering. In all, there are 9^3 placement possibilities, and he stores that as a bit vector called $removed (in his article, although in the full code, it's a hash key named removed). Each bit is a candidate row, although if he sets a bit, that row is no longer a candidate. The index of that bit maps into an array he keeps elsewhere. By turning of rows in his bitmask, he doesn't have to remove elements from the middle of his data structure, saving him a lot of time Perl would otherwise spend dealing with data structure maintenance.

Once he knows that he's going to skip a row, he sets that bit in the $removed bit vector.

```
vec( $removed, $row, 1 ) = 1;
```

When he needs to know all of the candidate rows still left, that's just the bitwise negation of the removed rows. Be careful here! You don't want the binding operator by mistake.

```
$live_rows = ( ~ $removed );
```

Summary

Now you have the basics of bit vectors, including the basic operations to play with bits, store them with vec, and save yourself a lot of memory. That savings comes at the cost of some programming complexity,

About the Author

brian d foy is the publisher of *The Perl Review*.