Finding the shortest binary string in a given interval

devblogs.microsoft.com/oldnewthing/20160222-00

February 22, 2016

Raymond Chen

Today's Little Program answers the question, "Given an interval of real numbers, find the shortest binary string that fits in that interval." For example, if you are given the range 0.1… 0.3, then the answer is 0.25, whose binary representation is 0.01. Another way of phrasing the problem is that you are to find the simplest dyadic rational number in the range, where a *dyadic* rational number is one whose denominator is a power of two. In our case, the simplest such fraction is ¼.

This sounds like a pointless exercise, but it actually solves a real problem: The shortest binary string in the interval (a, b) is the value of the surreal number $\{ a \mid b \}$.[1]

The idea here is that we start with an integer candidate, and the refine the candidate, making it longer and longer by one bit (alternatively: doubling the denominator), until we come up with a candidate that fits between a and b. Here is my first attempt:

```
function nextHigherInteger(a) {
 var t = Math.ceil(a);
 if (t == a) t++;
 return t;
}

// Preconditions: 0 ≤ a < b
function shortest(a, b) {
 var t = nextHigherInteger(a);
 var bit = 1;
 while (t >= b) {
  bit /= 2;
  t -= bit;
 }
 return t;
}
```

You may have noticed that the code is italicized. That's because it's wrong, but let's walk through it anyway so we can understand what it is trying to do:

We start by calculating the smallest integer greater than or equal to *a*. That is our candidate. If that is less than *b*, then we have a winner. Otherwise, we subtract a bit and try again. Each time we try, the amount we subtract is cut in half, so that it remains a single bit.

This seems to work for `shortest(0.1, 0.3)`, but it doesn't work for `shortest(0.3, 0.4)` because we forgot to check whether subtracting the next bit took us below *a* and out of the range.

Let's try again:

```
function shortest(a, b) {
 var t = nextHigherInteger(a);
 var bit = 1;
 while (bit) {
  bit /= 2;
  if (t <= a) t += bit;
  else if (t >= b) t-= bit;
  else return t;
 }
 throw "Unable to find a value in range";
}
```

This time, after making our initial guess, we see if it lies in the target range. If it's too low, we bump it up a little; if it's too high, we drop it a little. Each time we adjust the value, we do so by a smaller and smaller amount, which lets us hit a smaller and smaller target range. The total adjustment is the Zeno sum $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots \to 1$. As a worst case, our initial guess overshoots *a* by $1 - \varepsilon$, so our repeated adjustments will eventually bring us as close to *a* as desired, if you just wait long enough. Since we assume that $b > a$, we will eventually adjust enough so that the adjusted value is between *a* and *b*.

Another way of looking at this problem is by zooming the range up rather than zooming the trial value down; *i.e.*, we keep shifting the binary point to the right (doubling the size of the range) until we can fit an integer into the range. Once we find such an integer, we can then scale it back down to the original interval.

```
function shortest(a, b) {
 var den = 1;
 var t = nextHigherInteger(a);
 while (t >= b) {
  a *= 2;
  b *= 2;
  den *= 2;
  t = nextHigherInteger(a);
 }
 return t / den;
}
```

[1] Therefore, it doesn't so much solve a real problem as it solves a surreal problem. Yes, it's a bad math pun. I don't apologize.

Raymond Chen

Follow