28. Algorithms

By: Steve Krenzel

Previous Index Next

Deriving Newton's Method from Binary Search

Newton's method is a popular algorithm for calculating the square root of a number, however, it's easy to forget how to implement it. We can fix that by taking an algorithm that any computer scientist should know, the binary search algorithm, and following a few simple transformations to derive an implementation of Newton's method.

First, here's the binary search code in python to calculate the square root of a number:

```
def sqrt(n, precision=10e-8):
  lo, hi = 0.0, max(n, 1.0)
  prev, mid = 0, (lo + hi) / 2.0
  while abs(mid - prev) > precision:
    if mid ** 2 > n:
      hi = mid
    else:
      lo = mid
    prev, mid = mid, (lo + hi) / 2.0
  return mid
```

All we're doing here is binary searching through the range of **0** to **N** to find the square root of **N**. It's a pretty standard binary search where we calculate the midpoint of the range and then set the midpoint to either the upper or lower bound of the range depending upon whether the midpoint squared is greater than or lesser than **N** (respectively).

The first observation that we must make is that if we divide **N** by a number

greater than it's square root, we get a number that is less than it's square root. Conversely, if we divide **N** by a number less than it's square root we get a number greater than it's square root. And of course, dividing **N** by it's square root gives us it's square root. In other words:

sqrt(49) = 7
49 / 9 = 5.444
49 / 3 = 16.333
49 / 7 = 7

Given this information, we can actually update both the lower and upper bounds on each iteration, instead of just updating one. If we calculate an upper bound, then we can also calculate a new lower bound by dividing **N** by the upper bound. Likewise, if we calculate a lower bound, then we can also calculate a new upper bound by dividing **N** by the lower bound. The code looks like:

```
def sqrt(n, precision=10e-8):
  lo, hi = 0.0, max(n, 1.0)
  prev, mid = 0, (lo + hi) / 2.0
  while abs(mid - prev) > precision:
    if mid ** 2 > n:
      hi = mid
      lo = n / mid    # <---- We add this line
    else:
      lo = mid
      hi = n / mid    # <---- and this line
    prev, mid = mid, (lo + hi) / 2.0
  return mid
```

Now we observe that we calculate the midpoint by taking the average of the upper and lower bounds. In one case, we calculate:

(hi + lo) / 2.0 = (mid + (n / mid)) / 2.0

and in the other case we calculate

$$(lo + hi) / 2.0 = (mid + (n / mid)) / 2.0$$

It turns out that either way, we're calculating the same value! Now we can simplify our code to:

```
def sqrt(n, precision=10e-8):
  lo, hi = 0.0, max(n, 1.0)
  prev, mid = 0, (lo + hi) / 2.0
  while abs(mid - prev) > precision:
    prev, mid = mid, (mid + (n / mid)) / 2.0
  return mid
```

We no longer need to track an upper and lower bound, so we can get rid of that first line as well. We wind up with:

```
def sqrt(n, precision=10e-8):
  prev, mid = 0, float(n)
  while abs(mid - prev) > precision:
    prev, mid = mid, (mid + (n / mid)) / 2.0
  return mid
```

And that's it. We've derived Newton's method from a binary search.

About the author
I'm Steve Krenzel, a software engineer and co-founder of Thinkfuse.
Feel free to send me an email at sgk284@gmail.com

Previous Next