Computing Logarithm Bit-by-Bit

Nitin Verma
mathsanew.com

August 4, 2021

Given any positive real number a, we want to find its logarithm in base 2, i.e. $\log_2 a$. In this article, whenever we specify "log", it will mean $\log_2$. Say, $\log a = n + f$, where n is an integer and f is a fraction with $0 \leq f < 1$. That is, $a = 2^{n+f} = 2^n 2^f$, where $1 \leq 2^f < 2$. Note that n is negative when $a < 1$. Also, $2^n \leq a < 2^{n+1}$ for all a.

In this article, we will often refer to the binary representation of some non-negative real number r, denoted as: $(r_m r_{m-1} \ldots r_0.r_{-1}r_{-2}\ldots)_2$. The bit r_i is said to be at *position* i. r_m is the most-significant-bit (MSB). The *radix-point* (.) separates the integer and fractional part of r. The value of r is given by:

$$r_m(2^m) + r_{m-1}(2^{m-1}) + \ldots + r_0(2^0) + r_{-1}\left(\frac{1}{2^1}\right) + r_{-2}\left(\frac{1}{2^2}\right) + \ldots$$

For example, in $(110.101)_2$, $(110)_2$ represents 6 and $(0.101)_2$ represents $5/8 = 0.625$, giving the complete value of 6.625 in decimal.

If a is an integer and stored as an integer type, we can check the position of its MSB in the binary form. This bit position gives us the value of n. If a is a real number stored as a floating-point of IEEE standard 754 ($significand \times 2^{exponent}$), again we can exploit the storage format to find n.

If we do not want to depend upon the storage format of a, we can find n by other methods too. For example, if $a \geq 2$, we can iteratively divide it by 2 until $a < 2$, while counting the number of divisions needed (the count is n). Similarly, if $a < 1$, we can iteratively multiply it by 2 until $a \geq 1$ (the count is $(-n)$). There can be other more efficient methods also.

We now come to the problem of finding f. Once we know n, we will divide a by 2^n to give us another real number $x = a/2^n = 2^f$, where $1 \leq x < 2$. We need to find $f = \log x$. There are a few algorithms known for this, and this operation is sometimes also provided by the hardware itself. We will discuss a simple algorithm which is based on very basic mathematics. It finds f iteratively one bit at a time.

Bit-by-Bit Algorithm

The algorithm may have been discovered multiple times independently, but appears to be first published in 1956 by D. R. Morrison [1]. Morrison provided a generic algorithm to find inverse of any function which meets certain conditions, 2^y being one such function. Note, $\log_2(2^y) = y$, so the inverse of 2^y is the $\log_2$ function.

Since $0 \leq f < 1$, it can be written in binary form as: $(0.b_1b_2b_3\ldots)_2$, where each b_i is a bit (0 or 1) at position $(-i)$. The algorithm makes use of the below observations:

$$\begin{aligned}
& & f &= \log x & (1)\\
\Leftrightarrow & & 2f &= \log(x^2) \\
& & & \{\text{since } 2f = 2(0.b_1b_2b_3\ldots)_2 = (b_1.b_2b_3\ldots)_2\} \\
\Leftrightarrow & & (b_1.b_2b_3\ldots)_2 &= \log(x^2)
\end{aligned}$$

So, $x^2 \geq 2$ implies b_1 must be 1, and $x^2 < 2$ implies b_1 must be 0. Thus, b_1 can be deduced just by comparing x^2 with 2. Further,

$$\begin{aligned}
& & (b_1.b_2b_3\ldots)_2 &= \log(x^2) \\
\Leftrightarrow & & (0.b_2b_3\ldots)_2 &= \log(x^2) - (b_1)_2 \\
& & &= \log\left(\frac{x^2}{2^{b_1}}\right) \\
& & &= \log(x^2) \text{ or } \log\left(\frac{x^2}{2}\right) \quad \{\text{for } b_1 = 0 \text{ or } 1 \text{ respectively}\}
\end{aligned}$$

If we now consider $(0.b_2b_3\ldots)_2$ to be our new f, and x^2 or $x^2/2$ (based on b_1) to be our new x, then above relation translates to $f = \log x$, which is same as (1). Thus, bit b_2 can be found by the same process we used to find b_1. Repeating this process m times will give us m bits of f up to b_m, which we can combine to give us an approximation for f as $(0.b_1b_2b_3\ldots b_m)_2$.

Below is an implementation of this algorithm in C. Input x is assumed to follow $1 \leq x < 2$. m specifies the number of bits to generate for f.

```
/* base-2 log bit-by-bit */
float log2_bbb(float x, int m)
{
  int i, bits, bit;

  if(x == 1)
    return 0;

  i = 0;
  bits = 0;

  /* loop-invariants:
       P: f = log(x) (variable f is unknown and shown commented)
       Q: 1 <= x < 2
       R: 'bits' contains i bits appended as 'bit' */

  while(i < m)
  {
    /* f <-- 2 * f */    /* maintain P */
    x = x * x;

    if(x >= 2)
    {
      bit = 1;
      /* f <-- f - 1 */    /* maintain P */
      x = x/2;
    }
    else
      bit = 0;

    bits = (bits << 1) | bit;

    i++;
  }

  /* 'bits' contains m bits from f, so f =(approx) bits/(2^m) */

  return ((float)bits) / (1 << m);
}
```

There can be other variants of this algorithm. For example, we can write it for computing logarithm in any base b ($\log_b x$). For that, the comparison $x^2 \geq 2$ needs to be replaced by $x^2 \geq b$ to deduce the bit. And for the new x, division $x^2/2$ needs to be replaced by x^2/b.

Another variant is to treat f in any other radix d as $(0.d_1d_2d_3\ldots)_d$, where each d_i is a digit in radix d. Then, instead of generating f one bit at a time, we can generate it one (radix-d) digit at a time. For that, both sides of (1) are multiplied by d, and instead of x^2 we need to compute x^d. Then, we need to find digit d_1 such that $2^{d_1} \leq x^d < 2^{d_1+1}$. After finding d_1, we will compute $x^d/2^{d_1}$ instead of $x^2/2^{b_1}$ for our new x.

Reversing an Exponentiation Algorithm

Since $f = \log_2 x$ is equivalent to $2^f = x$, we can say that finding logarithm is reverse of doing exponentiation. In an earlier article titled *Some Algorithms for Exponentiation* [2], we discussed Binary (Left-to-Right) algorithm for computing a^b where a and b are positive integers.

Can we create a similar algorithm to compute 2^f where f is not an integer, but a real number with $0 \leq f < 1$. That is, can we iteratively process the bits of $f = (0.b_1b_2b_3\ldots)_2$ to build-up 2^f, as we did in the Binary (Left-to-Right) algorithm to build-up a^b? Below we see how this can be done; note that it processes bits of f in right-to-left order:

```
float binary_exp(float f)
{
  float x, y;
  int m = 8, p, bit = 0;

  /* say f = (0.b{-1} b{-2} b{-3} ... b{-m}) in binary (m bits) */

  p = -(m+1);
  y = 0;
  x = 1;

  /* loop-invariants:
       Y: y consists of suffix bits of f from position p to -m;
          thus y = 0.(bp b[p-1] ... b{-m}) (implies 0 <= y < 1).
       X: x = 2^y (implies 1 <= x < 2) */

  while(p < -1)
  {
    p = p + 1;

    /* bit = {read the next bit of f, from position p} */

    /* maintain Y */
    y = (y + bit)/2;

    /* maintain X */
    if(bit)
      x = x * 2;

    x = sqrtf(x); /* sqrtf() computes the square-root */
  }
  /* p = -1 */
  /* y = f */
  /* x = 2^f */

  return x;
}
```

It need not be an efficient way to compute 2^f. But we wish to see if this exponentiation algorithm, where f is given and 2^f is computed, can help us create an algorithm to compute f if 2^f is given. Can we reverse the execution of the loop in this method so that we can start with $x = 2^f$ (x known, f unknown) and generate the bits of f one by one?

Suppose we know x at the end of an iteration, call it x_e. Note that each iteration maintains $1 \leq x < 2$. The value of x at the start of this iteration must be $x_s = x_e^2$ or $x_e^2/2$ depending upon the bit processed. But x_s too must follow $1 \leq x_s < 2$. So, $x_s = x_e^2$ iff $1 \leq x_e^2 < 2$. And $x_s = x_e^2/2$ iff

$1 \leq x_e^2/2 < 2$, i.e. $2 \leq x_e^2 < 4$. So, x_s and hence the bit can be deduced based on whether $x_e^2 \geq 2$ or not.

Thus we have found a way to reach at the starting state of an iteration from its end; to reverse the execution of an iteration. Starting with the final value of $x(= 2^f)$, we can now execute all iterations in the reverse order, with each iteration itself reversed as above. Below we see the loop effecting such reversal:

```
p = -1;
/* variable y is not shown */
/* x = 2^f is given, 1 <= x < 2 */

while(p > -(m+1))
{
  x = x * x;

  if(x >= 2)
  {
    bit = 1;
    x = x/2;
  }
  else
    bit = 0;

  /* 'bit' is the bit of f at position p */

  p = p - 1;
}
/* p = -(m+1) */
```

Thus we could find a method to generate the bits of f one by one. Notice that this algorithm is same as the Bit-by-Bit algorithm we discussed earlier, but we have arrived at it by reversing an exponentiation algorithm.

■

References

[1] D. R. Morrison. *A Method for Computing Certain Inverse Functions.* Math. Comp. (AMS), Vol 10 (1956), 202–208. https://www.ams.org/journals/mcom/1956-10-056/S0025-5718-1956-0083821-9.

[2] Nitin Verma. *Some Algorithms for Exponentiation.* https://mathsanew.com/articles/algorithms_for_exponentiation.pdf (2021).