Integer Logarithm
Aubrey Jaffer
March 2008

Bignum software packages provide operations on practically unbounded size integers. They typically include an integer-exponentiation operator but not its inverse:

procedure: integer-log base j

Returns the largest integer whose power of positive integer base is less than or equal to positive integer j.

Searching the web for “integer logarithm” finds uses of integer logarithms, the non-power-of-two logarithms usually computed using floating-point arithmetic. But I didn’t find an integer algorithm with time-complexity better than $O(n^3)$, where $n$ is the number of digits.

The approach here is to minimize the number of divisions while knocking down the size of input $j$ as quickly as possible, but without generating intermediate numbers larger than $j$. Repeatedly squaring the base provides the exponentially increasing divisors. An internal function ilog calls itself with exponentially growing $b$ and exponentially shrinking $k/b$ until $b > k$. Each call then divides the returned ilog value by its $b$ if doings so does not result in 0.

This Scheme function employs only integer operations:

```
(define (integer-log base j)
  (define n 1)
  (define (ilog m b k)
    (cond ((> b k) k)
      (else (set! n (+ n m))
        (let ((q (ilog (* 2 m) (* b b) (quotient k b))))
          (cond ((> b q) q)
            (else (set! n (+ n m))
              (quotient q b)))))))
  (cond ((> base j) 0)
    (else (ilog 1 base (quotient j base))
      n))))
```

For $j = base^p + c, c < base^p$:

- $m_0 = 1$
- $b_0 = base$
- $k_0 = floor(j/base)$
- $m_1 = 2$
- $b_1 = base^2$
- $k_1 = floor(j/base^2)$
- $m_2 = 4$
- $b_2 = base^4$
- $k_2 = floor(j/base^4)$
- $m_L = 2^L$
- $b_L = base^{(2^L)}$
- $k_L = floor(j/base^{(2^L)})$

The variable $n$ accumulates all the $m_i$ values for calls where $k_i \geq b_i$. During the $L$th call, where $k_L \geq b_L$:

$$n = 2^{(L+1)}$$
When \( k_L < b_L \), the most nested call returns \( k_L \) without altering \( n \). Then each stacked call compares the returned value \( q_L \) with \( b_L \); if greater, it adds \( m_L \) to \( n \) and returns \( q/b_L \); otherwise it merely returns \( q_L \).

Counting the number of base factors divided from \( j \), \( n \) accumulates between \( 2^{(L+1)} \) and \( 2^{(L+2)} - 1 \) (where \( L \) is the number of calls with \( k_L \geq b_L \)).

The largest \( b_L = \text{base}^{(2^{(L+1)})} \) generated in the calculation is passed to \( \text{ilog} \) where \( k_L < b_L \), which is not counted in \( n \). This largest \( b_L \) is always less than or equal to \( j = \text{base}^{(n)} + c = \text{base}^{(2^{(L+1)})} + c \).

The number of operations is logarithmic in \( p \), the number of digits of \( j \). In long-division, the time-complexity of dividing a \( n \)-digit number by a \( d \)-digit number is bounded by \( O((n - d) \cdot d) \) [KNUTH]. The first few divisions will dominate running time; the conditional divisions done on return have small \( n - d \).

Let \( p = 2^{H+1} \) be the number of digits in \( j \). The time-complexity of the long-divisions is proportional to:

\[
\sum_{L=0}^{H} ((p - 2^L) - 2^L)2^L = \sum_{L=0}^{H} (2^{H+1} - 2^{L+1})2^L = 2^{H+1} \sum_{L=0}^{H} 2^L - 2 \sum_{L=0}^{H} 2^{2L} = 2^{H+1} \cdot \frac{2^{H+1} - 1}{2 - 1} - 2 \cdot \frac{4^{H+1} - 1}{4 - 1} = p \cdot \frac{p - 1}{2 - 1} - 2 \cdot \frac{p^2 - 1}{4 - 1} = \frac{p^2 - 3p + 2}{3} < O(p^2)
\]

The time-complexity of the repeated squarings with \( O(n^2) \) multiplication is proportional to:

\[
\sum_{L=0}^{H} (2^L)^2 = \sum_{L=0}^{H} 4^L = \frac{4^{H+1} - 1}{4 - 1} = \frac{p^2 - 1}{3} < O(p^2)
\]

Thus the overall time-complexity when using long division is \( O(p^2) \).

**Bibliography**

[KNUTH] Donald E. Knuth.