Shifting *n*th root algorithm

From Wikipedia, the free encyclopedia
(Redirected from Shifting nth-root algorithm)

This article **does not cite any references or sources**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. *(May 2010)*

The **shifting *n*th root algorithm** is an algorithm for extracting the *n*th root of a positive real number which proceeds iteratively by shifting in *n* digits of the radicand, starting with the most significant, and produces one digit of the root on each iteration, in a manner similar to long division.

Contents [hide]

Algorithm [edit]

Notation [edit]

Let *B* be the base of the number system you are using, and *n* be the degree of the root to be extracted. Let *x* be the radicand processed thus far, *y* be the root extracted thus far, and *r* be the remainder. Let α be the next *n* digits of the radicand, and β be the next digit of the root. Let *x'* be the new value of *x* for the next iteration, *y'* be the new value of *y* for the next iteration, and *r'* be the new value of *r* for the next iteration. These are all integers.

Invariants [edit]

At each iteration, the invariant $y^n + r = x$ will hold. The invariant $(y+1)^n > x$ will hold. Thus *y* is the largest integer less than or equal to the *n*th root of *x*, and *r* is the remainder.

Initialization [edit]

The initial values of *x*, *y*, and *r* should be 0. The value of α for the first iteration should be the most significant aligned block of *n* digits of the radicand. An aligned block of *n* digits means a block of digits aligned so that the decimal point falls between blocks. For example, in 123.4 the most significant aligned block of 2 digits is 01, the next most significant is 23, and the third most significant is 40.

Main loop [edit]

On each iteration we shift in *n* digits of the radicand, so we have $x' = B^n x + \alpha$ and we produce 1 digit of the root, so we have $y' = By + \beta$. We want to choose β and *r'* so that the invariants described above hold. It turns out that there is always exactly one such choice, as will be proved below.

The first invariant says that:

$$x' = y'^n + r'$$

or

$$B^n x + \alpha = (By + \beta)^n + r'.$$

So, pick the largest integer β such that

$$(By + \beta)^n \le B^n x + \alpha$$

and let

$$r' = B^n x + \alpha - (By + \beta)^n.$$

Such a β always exists, since if $\beta = 0$ then the condition is $B^n y^n \le B^n x + \alpha$, but $y^n \le x$, so this is always true. Also, β must be less than *B*, since if $\beta = B$ then we would have

$$(B(y+1))^n \le B^n x + \alpha$$

but the second invariant implies that

$$B^n x < B^n (y+1)^n$$

and since $B^n x$ and $B^n (y+1)^n$ are both multiples of B^n the difference between them must be at least B^n, and then we have

$$B^n x + B^n \le B^n (y+1)^n$$
$$B^n x + B^n \le B^n x + \alpha$$
$$B^n \le \alpha$$

but $0 \le \alpha < B^n$ by definition of α, so this can't be true, and $(By + \beta)^n$ is a monotonically increasing function of β, so it can't be true for larger β either, so we conclude that there exists an integer γ with $\gamma < B$ such that an integer *r'* with $r' \ge 0$ exists such that the first invariant holds if and only if $0 \le \beta \le \gamma$.

Now consider the second invariant. It says:

$$(y' + 1)^n > x'$$

or

$$(By + \beta + 1)^n > B^n x + \alpha$$

Now, if β is not the largest admissible β for the first invariant as described above, then $\beta + 1$ is also admissible, and we have

$$(By + \beta + 1)^n \le B^n x + \alpha$$

This violates the second invariant, so to satisfy both invariants we must pick the largest β allowed by the first invariant. Thus we have proven the existence and uniqueness of β and *r'*.

To summarize, on each iteration:

1. Let α be the next aligned block of digits from the radicand
2. Let $x' = B^n x + \alpha$
3. Let β be the largest β such that $(By + \beta)^n \le B^n x + \alpha$
4. Let $y' = By + \beta$
5. Let $r' = x' - y'^n$

Now, note that $x = y^n + r$, so the condition

$$(By + \beta)^n \le B^n x + \alpha$$

is equivalent to

$$(By + \beta)^n - B^n y^n \le B^n r + \alpha$$

and

$$r' = x' - y'^n = B^n x + \alpha - (By + \beta)^n$$

is equivalent to

$$r' = B^n r + \alpha - ((By + \beta)^n - B^n y^n)$$

Thus, we don't actually need x, and since $r = x - y^n$ and $x < (y+1)^n$, $r < (y+1)^n - y^n$ or $r < ny^{n-1} + O(y^{n-2})$, or $r < nx^{\frac{n-1}{n}} + O(x^{\frac{n-2}{n}})$, so by using r instead of x we save time and space by a factor of 1/*n*. Also, the $B^n y^n$ we subtract in the new test cancels the one in $(By + \beta)^n$, so now the highest power of *y* we have to evaluate is y^{n-1} rather than y^n.

Summary [edit]

1. Initialize *r* and *y* to 0.
2. Repeat until desired precision is obtained:
   1. Let α be the next aligned block of digits from the radicand.
   2. Let β be the largest β such that $(By+\beta)^n - B^n y^n \le B^n r + \alpha.$
   3. Let $y' = By + \beta$.
   4. Let $r' = B^n r + \alpha - ((By+\beta)^n - B^n y^n).$
   5. Assign $y \leftarrow y'$ and $r \leftarrow r'.$
3. y is the largest integer such that $y^n < xB^k$, and $y^n + r = xB^k$, where k is the number of digits of the radicand after the decimal point that have been consumed (a negative number if the algorithm hasn't reached the decimal point yet).

Paper-and-pencil *n*th roots [edit]

As noted above, this algorithm is similar to long division, and it lends itself to the same notation:

```
      1.  4   4   2   2   4
    ----------------------
 _ 3/ 3.000 000 000 000 000
  \/  1                        = 3(10×0)^2×1      +3(10×0)×1^2      +1^3
      -
      2 000
      1 744                    = 3(10×1)^2×4      +3(10×1)×4^2      +4^3
      -----
        256 000
        241 984                = 3(10×14)^2×4     +3(10×14)×4^2     +4^3
        -------
         14 016 000
         12 458 888            = 3(10×144^2)×2    +3(10×144)×2^2    +2^3
         ----------
          1 557 112 000
          1 247 791 448        = 3(10×1442^2)×2   +3(10×1442)×2^2   +2^3
          -------------
            309 320 552 000
            249 599 823 424    = 3(10×14422^2)×4  +3(10×14422)×4^2  +4^3
            ---------------
             59 720 728 576
```

Note that after the first iteration or two the leading term dominates the $(By+\beta)^n - B^n y^n$, so we can get an often correct first guess at β by dividing $r + \alpha$ by $nB^{n-1}y^{n-1}$.

Performance [edit]

On each iteration, the most time-consuming task is to select β. We know that there are *B* possible values, so we can find β using $O(\log(B))$ comparisons. Each comparison will require evaluating $(By+\beta)^n - B^n y^n$. In the *k*th iteration, y has *k* digits, and the polynomial can be evaluated with $2n-4$ multiplications of up to $k(n-1)$ digits and $n-2$ additions of up to $k(n-1)$ digits, once we know the powers of *y* and β up through $n-1$ for *y* and *n* for β. β has a restricted range, so we can get the powers of β in constant time. We can get the powers of *y* with $n-2$ multiplications of up to $k(n-1)$ digits. Assuming *n*-digit multiplication takes time $O(n^2)$ and addition takes time $O(n)$, we take time $O(k^2 n^2)$ for each comparison, or time $O(k^2 n^2 \log(B))$ to pick β. The remainder of the algorithm is addition and subtraction that takes time $O(k)$, so each iteration takes $O(k^2 n^2 \log(B))$. For all *k* digits, we need time $O(k^3 n^2 \log(B))$.

The only internal storage needed is *r*, which is $O(k)$ digits on the *k*th iteration. That this algorithm doesn't have bounded memory usage puts an upper bound on the number of digits which can be computed mentally, unlike the more elementary algorithms of arithmetic. Unfortunately, any bounded memory state machine with periodic inputs can only produce periodic outputs, so there are no such algorithms which can compute irrational numbers from rational ones, and thus no bounded memory root extraction algorithms.

Note that increasing the base increases the time needed to pick β by a factor of $O(\log(B))$, but decreases the number of digits needed to achieve a given precision by the same factor, and since the algorithm is cubic time in the number of digits, increasing the base gives an overall speedup of $O(\log^2(B))$. When the base is larger than the radicand, the algorithm degenerates to binary search, so it follows that this algorithm is not useful for computing roots with a computer, as it is always outperformed by much simpler binary search, and has the same memory complexity.

Examples [edit]

Square root of 2 in binary [edit]

```
      1. 0  1  1  0  1
    ------------------
_  / 10.00 00 00 00 00     1
 \/   1                  + 1
     -----               ----
      1 00                100
         0               +  0
     --------            -----
      1 00 00             1001
        10 01            +   1
     -----------         ------
         1 11 00          10101
         1 01 01         +    1
         ----------      -------
            1 11 00       101100
                  0      +     0
            ----------   --------
            1 11 00 00    1011001
            1 01 10 01          1
            ----------
               1 01 11 remainder
```

Square root of 3 [edit]

```
     1. 7  3  2  0  5
    ----------------------
_  / 3.00 00 00 00 00
 \/  1 = 20×0×1+1^2
     -
     2 00
     1 89 = 20×1×7+7^2
     ----
       11 00
       10 29 = 20×17×3+3^2
       -----
          71 00
          69 24 = 20×173×2+2^2
          -----
           1 76 00
                 0 = 20×1732×0+0^2
           -------
           1 76 00 00
           1 73 20 25 = 20×17320×5+5^2
           ----------
              2 79 75
```

Cube root of 5 [edit]

```
     1.  7   0   9   9   7
    ----------------------
_ 3/ 5. 000 000 000 000 000
 \/  1 = 300×(0^2)×1+30×0×(1^2)+1^3
     -
     4 000
```

```
    3 913 = 300×(1^2)×7+30×1×(7^2)+7^3
    -----
       87 000
            0 = 300×(17^2)*0+30×17×(0^2)+0^3
      -------
       87 000 000
       78 443 829 = 300×(170^2)×9+30×170×(9^2)+9^3
       ----------
        8 556 171 000
        7 889 992 299 = 300×(1709^2)×9+30×1709×(9^2)+9^3
        -------------
          666 178 701 000
          614 014 317 973 = 300×(17099^2)×7+30×17099×(7^2)+7^3
          ---------------
           52 164 383 027
```

Fourth root of 7 [edit]

```
      1.   6    2    6    5    7
     ---------------------------
 _ 4/ 7.0000 0000 0000 0000 0000
  \/  1 = 4000×(0^3)×1+400×(0^2)×(1^2)+40×0×(1^3)+1^4
      -
      6 0000
      5 5536 = 4000×(1^3)×6+600×(1^2)×(6^2)+40×1×(6^3)+6^4
      ------
        4464 0000
        3338 7536 = 4000×(16^3)×2+600*(16^2)×(2^2)+40×16×(2^3)+2^4
        ---------
        1125 2464 0000
        1026 0494 3376 = 4000×(162^3)×6+600×(162^2)×(6^2)+40×162×(6^3)+6^4
        --------------
          99 1969 6624 0000
          86 0185 1379 0625 = 4000×(1626^3)×5+600×(1626^2)×(5^2)+
          -----------------   40×1626×(5^3)+5^4
          13 1784 5244 9375 0000
          12 0489 2414 6927 3201 = 4000×(16265^3)×7+600×(16265^2)×(7^2)+
          ----------------------   40×16265×(7^3)+7^4
           1 1295 2830 2447 6799
```

External links [edit]

- Why the square root algorithm works "Home School Math". Also related pages giving examples of the long-division-like pencil and paper method for square roots.

Categories: Root-finding algorithms | Computer arithmetic algorithms