**Square root bit-by-bit, or restorative algorithm**

This algorithm on every step finds precise value of the next digit of the square starting from the top digits and going to bottom digits. The algorithm can find roots with any required precision and is able to continue to find digits indefinitely.

Idea behind the algorithm:

Every number can be represented as a sum of its digits multiplied by base in appropriate power. For example:

Or in general we can write:

Note that here each is single digit raised to specific order of magnitude. When is a square root of a number we can write that:

If we apply square to the sum, we get following expression

Let’s look at a term of this expression .

Here is a number a defined up to top digit j with all following bottom digits set to 0. In other words, in each term this sum is progressively precise approximation of a. And in each term is a next digit that we need to define.

The algorithm is built on exploiting the structure of this term. Suppose we know j top digits already. To find next digit we go through all possible values for a digit from bigger to smaller. For each candidate value we compute up to current term. If is bigger than input value x that means our pick is too large and we check the next candidate value.

We call already computed resulting value “” if it is computed up (and including) to digit :

Note, that this value is for , and not for !

We denote the term that we are considering on current step when we try . This reveals recursive relation:

Also we can denote “” (remainder) digits we have not yet found starting and including digit . This remainder also includes term that we are considering on each step. This remainder is the difference between true value of and the approximation found so far.

Then we can write simple relation:

We can denote substitution of value into term expression as . Note again, that value is a single digit multiplied by appropriate base power.

Then on each iteration of the algorithm we find largest v that following inequality holds:

And we can rewrite this expression in terms of remainder:

This way we can look at each term individually without keeping track of the total sum that approximates . Each found term just reduces the remainder:

And finally, this inequality can be put in a form that relates it to :

You can see that the requirement not to exceed the remainder (or true square value) forces us to produce floor rounded approximation for the root. Our approximation approaches true value for from below. Also, digits of the approximation found on each step of this algorithm are exact and final digits that are present in the true value of on corresponding places.

In the case of the real numbers the last expression allows us to achieve any arbitrary precision by continuing to find digits. When decimal place of certain goes beyond decimal point we can continue by simply considering its decimal place negative. And if the true value of the root is irrational remainder can be reduced infinitely.

Now, let’s look at binary version of this algorithm. Here we are considering input number to be 64 bit integer and we assume that all 64 bits represent integer part. In other words there is no fractional part. The result of taking square root of 64-bit number will always fit into 32-bit number. So, the result is 32-bit number.

When considering binary numbers, a few tricks allow us to perform only power-of-two multiplications in form of cheap binary shifts and to avoid more expensive regular multiplications.

First, let’s show base multipliers explicitly by denoting:

When we check if the term fits in the remainder with certain value of in base-2 there are only two options for value of – 0 or , or simply . Therefore, we can check only . Each iteration that searches for the next term has only one candidate value to consider, we call this value . Candidate must be one digit binary number multiplied by some power of two. First candidate numbers in binary:

And in general:

On each iteration this number is shifted right by one and equals to the number appropriate on each step.

But we can represent iteration in more complicated way to save computation on multiplications.

First, result buffer is 64-bit number instead of 32 bit. We do not fill it from 31st bit down to bit 0. We start filling it from 62rd bit and on each iteration, we write next bit and shift the whole result right. In essence we use :

and this also implies

Shifting of the result happens like this:

It is clear that after 32 iterations result will have width of 32 bits and will contain normal 32 bits value for the root of given number.

In addition to that we also represent iterator as 64-bit number starting with the second to highest bit. We define:

and this implies

On each iteration candidate number is shifted right twice to match corresponding position in the buffer:

If we remember that is just a power of two we can get:

So, with these definitions we can rewrite the formula for the iteration:

This way on each iteration instead of multiplication we can shift and once each and add them up.

One other improvement might be gained when we count towards res. Since is a binary number that has a single 1 in it that is aligned with appropriate bit in instead of addition we can use binary OR that is performed much faster.

The final code is:

static inline uint64\_t sqrt\_64(uint64\_t x) {

// Here we use x as the remainder variable

uint64\_t res = 0;

uint64\_t candidate = 0x4000000000000000; // 0100 0000 0000 0000 ...

uint64\_t term = 0;

// Skipping leading zeroes in x

while (candidate > x)

candidate >>= 2;

// Iterating

while (candidate != 0) {

// Next term to check

term = res | candidate;

if (term <= x) {

// Reducing the remainder

x -= term;

// Counting the result

res = (res >> 1)| candidate;

}

else

res >>= 1;

// Advancing to the next iteration

candidate >>= 2;

}

return res;

}

