Reversible Computing Case Study: Finding the Square Root of an Integer

Melissa A. Yu Kalyan S. Perumalla

August 8, 2013

1 Abstract

Here we consider a case study in reversible computing, namely, how to reversibly compute the integer square root $\lfloor\sqrt{x}\rfloor$ of an integer x. Starting with a simple linear time algorithm, we show how a simple reversal technique can be used to avoid saving the original number x in order to recover it. This reversal technique is shown to require half the memory that would otherwise be needed to save x. The linear time algorithm is then refined to improve the computation time to be only logarithmically dependent on the input size, giving a $O(\log x)$ run time using $\frac{1}{2}(1 + \lfloor\log_2 x\rfloor)$ bits of memory.

2 A Simple Square Root Algorithm

We would like to compute $y = \sqrt{x}$, where x and y are non-negative integers, i.e., $y = \lfloor\sqrt{x}\rfloor$. A simple linear algorithm (linear with respect to the square root value) for acomplishing this task involves testing successive y values until $y^2 \geq x$, i.e., until $dy = x - y^2 \leq 0$:

y=1 $dy_1 = x - 1^2$
y=2 $dy_2 = x - 2^2$
...
y=n $dy_n = x - n^2 \leq 0$

At this point, y can be found:

$$y = \sqrt{x} = \begin{cases} n & \text{if } dy_n = 0 \\ n-1 & \text{if } dy_n < 0 \end{cases}$$

The linear time algorithm is shown in Algorithm 1.

Algorithm 1 Linear Algorithm for Finding the Square Root

$y \leftarrow 1$
$x \leftarrow$ non-negative integer input
while $y^2 < x$ **do**
    $y \leftarrow y + 1$
end while
if $y^2 > x$ **then**
    $y \leftarrow y - 1$
end if
return y

3 Reversible Square Root Algorithm

3.1 Two Approaches

Consider the reversibilty of the linear algorithm presented earlier. We must recover the input x from the output n. Let us examine two methods of doing this:

- Memory-copying approach: Save x in addition to n at the end of the computation.
- Reversible computing approach: Because $n = \sqrt{x}$, then $x = n^2$. However, $dy_n = x - n^2$ is not necessarily 0. Thus, for accurate recovery of x, save dy_n along with n at the end of the computation.

3.2 Analysis of Memory Usage

Comparing these methods, we find that the main diference between them is that one encodes x while the other encodes dy_n at the end of the calculation. Therefore, we can compare the two by comparing the number of bits needed to encode dy_n versus x.

If: $n^2 \leq x < (n+1)^2$, i.e., $n^2 \leq x \leq (n+1)^2 - 1$

Then:

$$\begin{aligned} dy_n &= x - n^2 \\ &\leq [(n+1)^2 - 1] - n^2 \\ &\leq 2n \end{aligned}$$

The number of bits needed to encode dy_n and x ($|dy_n|$ and $|x|$ respectively) is:

$$\begin{aligned} |dy_n| &= \lfloor \log_2 dy_n \rfloor + 1 \\ &\leq \lfloor \log_2 2n \rfloor + 1 \\ &\leq \lfloor \log_2 n \rfloor + 2 \end{aligned} \qquad \begin{aligned} |x| &= \lfloor \log_2 x \rfloor + 1 \\ &\geq \lfloor \log_2 n^2 \rfloor + 1 \\ &\geq \lfloor 2 \log_2 n \rfloor + 1 \end{aligned}$$

Manipulating the above inequalities, we obtain the ratio of the number of bits needed to encode x versus dy_n:

$$\frac{|x|}{|dy_n|} = \frac{\lfloor \log_2 x \rfloor}{\lfloor \log_2 dy_n \rfloor}$$
$$\geq \frac{\lfloor 2\log_2 n \rfloor + 1}{\lfloor \log_2 n \rfloor + 2}$$
$$\geq 2$$

We can conlude that remembering dy_n instead of x saves memory by a minimum factor of about 2. This shows that the reversible computing approach is twice as memory-efficient as the memory-copying approach.

Through this simple example, we have seen that the challenge in developing algorithms for Reversible Computing is not in making them reversible, but in making them memory-efficient–any algorithm can be made reversible by simply adding a line to save the inputs at the end of the computation.

3.3 The Reversible Computing Algorithm

Now, let us consider how we might alter the basic forward-only square root algorithm (procedure "P") presented earlier to make it reversible using the Reversible Computing Approach.

Input	x
Procedure	P
Output	n

(a) Irreversible Forward, F_{orig}

Input	x
Procedure	P
Output	n, dy_n

(b) Reversible Forward, F_{rev}

Input	n, dy_n
Procedure	Q
Output	x

(c) Reverse, R_{rev}

As shown, the irreversible and reversible foward algorithms are virtually identical, using the same procedure P to carry out the calculation. However, the reversible foward stores dy_n in addition to n at the end of the calculation. In order to recover the input x from the output dy_n, we must define a new procedure Q: $x = n^2 + dy_n$.

3.4 Optimizing the Run-time of the Algorithm

The simple linear search algorithm we have used until now has been sufficient for us to see the difficulty of developing a memory-efficient reversible algorithm. However, this algorithm still needs to be optimized for speed. For large values of x, the algorithm would take a long time.

Consider the variant shown in Algorithm 2 to speed up our search which uses a "doubling approach" to save time by finding the general range of the square root before fine tuning. To quantify the savings in computation time, we will start with the observation

Algorithm 2 Logarithmic Algorithm for Finding the Square Root

```
k ← 0
x ← positive integer input
while 2^{2k} ≤ x do                    ▷ Keep doubling y until y^2 > x to find range for n
    k ← k + 1
end while
k ← k − 1
y ← 2^k                                ▷ n must be somewhere between 2^k and 2^{k+1}
------------------------------------------------------------
while k > 0 do                         ▷ Search within range using doubling approach again
    k ← k − 1
    if [y + 2^k]^2 ≤ x then
        y ← y + 2^k
    end if
end while
return y
```

that computation time is directly proprtional to the number of steps, and use this to find the computation time for the doubling approach.

$$T_{doubling} = \overbrace{\underbrace{T_1}_{\text{Step 1 Time}} + \underbrace{T_2}_{\text{Step 2 Time}}}^{\text{Total Doubling Approach Time}}$$

$$T_1 = O(\log \sqrt{x})$$
$$T_2 = O\left(\log \frac{\sqrt{x}}{2}\right)$$

$$T_{doubling} = O(\log \sqrt{x}) + O\left(\log \frac{\sqrt{x}}{2}\right) = O(\log \sqrt{x})$$

Applying the same reasoning, we find the time taken by our original algorithm (T_{orig}):

$$T_{orig} = O(\lfloor \sqrt{x} \rfloor).$$

4 Summary of Findings

Given an integer input x that is w bits long, $\lfloor \sqrt{x} \rfloor$ can be found *reversibly* using $O(\log \sqrt{x})$ steps and $\frac{w}{2}$ bits of memory.