

Exercise 1.45

We saw in section 1.3.3 that attempting to compute square roots by naively finding a fixed point of $y \mapsto x/y$ does not converge, and that this can be fixed by average damping. The same method works for finding cube roots as fixed points of the average-damped $y \mapsto x/y^2$. Unfortunately, the process does not work for fourth roots—a single average damp is not enough to make a fixed-point search for $y \mapsto x/y^3$ converge. On the other hand, if we average damp twice (i.e., use the average damp of the average damp of $y \mapsto x/y^3$) the fixed-point search does converge. Do some experiments to determine how many average damps are required to compute n th roots as a fixed-point search based upon repeated average damping of $y \mapsto x/y^{n-1}$. Use this to implement a simple procedure for computing n th roots using `fixed-point`, `average-damp`, and the `repeated` procedure of exercise 1.43. Assume that any arithmetic operations you need are available as primitives.

Answer¹

Foreword

In the following contexts, we will leave out many implementation of basic arithmetic operation, as the problem presentation recommends, so that we are able to focus on the key to the solution. The code and results presented here has been tested in MIT/GNU Scheme 9.1.1 on a MacBook Pro running OS X 10.8.


The `fixed-point-of-transform` procedure is so crucial that we have to present it here at the very beginning of this solution.

```
(define (fixed-point-of-transform g transform guess)
  (fixed-point (transform g) guess))
```

I Exploration

This section attempts to bring the underlying general pattern relates the n th root and the fixed-point search to the surface. To do this, we have to raise the quantity of the repeated average damping of $y \mapsto x/y^{n-1}$ increasingly and analyze the results generated in the meantime.

Starting here, we have to average damp twice for fourth roots, the problem description had made it quite clear.

¹Creative Commons  2012–2013 Lawrence R. Amlord (Shi-min Yan). This solution was originally created on Dec 21, 2012. Currently, this version was revised on May 5th, 2013.

```
(define (fourth-root x)
  (fixed-point-of-transform (lambda (y) (/ x (cube y)))
    (repeated average-damp 2)
    1.0))
```

Test:

```
> (fourth-root (* 6 6 6 6))
Value: 6.0
```

```
> (fourth-root (* 999 999 999 999))
Value: 999.0
```

```
> (fourth-root (* 10001 10001 10001 10001))
Value: 10001.0
```

It bears out the problem description. Now we want to know whether average damp twice works for the fifth roots, and further more, what is the highest roots order does it still works validly. Let's go on to write the definition of fifth root:

```
(define (fifth-root x)
  (fixed-point-of-transform (lambda (y) (/ x (expt y 4)))
    (repeated average-damp 2)
    1.0))
```

where the arithmetic operation `expt` here is defined in section 1.2.4. Try it:

```
> (fifth-root (expt 4 5))
Value: 4.000000066676982
```

```
> (fifth-root (expt 299 5))
Value: 298.9999999288808
```

```
> (fifth-root (expt 1099 5))
Value: 1099.0000000889913
```

So twice average damp also works fine with fifth root. Let's take see if it works well with the sixth roots:

```
(define (sixth-root x)
```

```
(fixed-point-of-transform (lambda (y) (/ x (expt y 5)))  
                          (repeated average-damp 2)  
                          1.0))
```

Here we have...

```
> (sixth-root (expt 3 6))  
Value: 2.999999799117626
```

```
> (sixth-root (expt 89 6))  
Value: 89.00000021985286
```

```
> (sixth-root (expt 2012 6))  
Value: 2012.0000001764565
```

So it does work with some particular integers as before. What about the seventh roots?

```
(define (seventh-root x)  
  (fixed-point-of-transform (lambda (y) (/ x (expt y 6)))  
                            (repeated average-damp 2)  
                            1.0))
```

Let's see...

```
> (seventh-root (expt 5 7))  
Value: 5.00000042720795
```

```
> (seventh-root (expt 98 7))  
Value: 98.00000040711095
```

```
> (seventh-root (expt 1999 7))  
Value: 1999.0000003393989
```

Go straight along the boring routine, let's examine the eighth root with twice average damping:

```
(define (eighth-root x)  
  (fixed-point-of-transform (lambda (y) (/ x (expt y 7)))  
                            (repeated average-damp 2)  
                            1.0))
```

It is this time, the eighth root, that makes the twice average damping strategy failed. If we sent the following expression to the Scheme interpreter, we will find that this program never terminates:

```
> (eighth-root (expt 100 8))
```

By increasing the times of average damping, we figure out that 3 times average damp works fine for eighth root.

```
(define (eighth-root x)
  (fixed-point-of-transform (lambda (y) (/ x (expt y 7)))
    (repeated average-damp 3)
    1.0))
```

```
> (eighth-root (expt 100 8))
Value: 100.0
```

```
> (eighth-root (expt 365 8))
Value: 365.0
```

```
> (eighth-root (expt 1001 8))
Value: 1001.0
```

Now Let see how far does the 3 times average damp extends to work, and try to clarify then underlying general pattern. Note that the definition of n th-root is almost identical except the exponent of y , so here we can leave them out to make out statement more concise:

```
> (nineth-root (expt 16 9))
Value: 15.999999962497867
```

```
> (tenth-root (expt 32 10))
Value: 32.00000012830529
```

```
> (eleventh-root (expt 64 11))
Value: 64.00000016042839
```

```
> (twelveth-root (expt 128 12))
Value: 127.99999971700788
```

```
> (thirteenth-root (expt 256 13))  
Value: 256.0000002471129
```

```
> (fourteenth-root (expt 512 14))  
Value: 511.99999964597305
```

```
> (fifteenth-root (expt 1024 15))  
Value: 1024.0000004256258
```

Once again, the interpreter hits an infinite loop while the following expression:

```
> (sixteenth-root (expt 2048 16))
```

Some one might guess that the **sixteenth-root** will probably work if we fixed the routine with one more average damp, let's see...

```
(define (sixteenth-root x)  
  (fixed-point-of-transform (lambda (y) (/ x (expt y 15)))  
    (repeated average-damp 4)  
    1.0))
```

```
> (sixteenth-root (expt 2048 16))  
Value: 2048.0
```

It really works!

II Induction

The presence of such a regular pattern is strong evidence that there is a useful underlying abstraction and we feel obliged to bring it to the surface.

If we pay attention to the number of average damps and the order of roots where they are valid to converge. We can summarize the patterns between them as the following:

Average Damps	Converged Root
1	1 ~ 3
2	4 ~ 7
3	8 ~ 15
4	16 ~ ...

Obviously, we need $\lfloor \log_2 n \rfloor$ average damps to converge to the n th root. we can check our guess out by increasing the number of average damps to 5 and see if it works for the 40th root:

```
(define (fortyth-root x)
  (fixed-point-of-transform (lambda (y) (/ x (expt y 39)))
    (repeated average-damp 5)
    1.0))
```

```
> (sixteenth-root (expt 95 40))
Value: 94.99999994278059
```

It did bear our prediction out.

So far, we have find out the underlying general pattern between the average damps and their converged roots. But we still have to implement the expression $\lfloor \log_2 n \rfloor$ in Lisp. Note that `floor` is a primitive operation in Scheme and $\lfloor \log_2 n \rfloor$ can be calculated in the following way mathematically:

$$\log_2 n = \frac{\ln n}{\ln 2}$$

where the mathematical operation $\ln x$ here can be implement with Lisp primitive operation `(log x)`. Just implement it in Lisp:

```
(define (floor-log-2 n)
  (floor (/ (log n) (log 2))))
```

With all the previous endeavor we have devoted, we can readily write the `nth-root` procedure as follow:

```
(define (nth-root x n)
  (let ((damp-times (floor-log-2 n)))
    (fixed-point-of-transform (lambda (y) (/ x (expt y (- n 1))))
      (repeated average-damp damp-times)
      1.0)))
```

We are going to end up with this problem with a few test with this general `nth-root` procedure:

```
> (nth-root (expt 9 7) 7)
Value: 9.000000335802724
```

```
> (nth-root (expt 89 24) 24)  
Value: 88.99999980123344
```

```
> (nth-root (expt 12 60) 60)  
Value: 12.00000044150704
```