

Authors: Kedar Mhaswade and Apoorv Mhaswade.  
Date Started: 15 November 2022.

## Introduction

This eponymous course on computational thinking has been hailed by many foremost computer experts and computer scientists. We refer you to Kai Wu's [comprehensive post](#) (sometimes inaccessible – but we have the [Wayback Machine](#)!) which talks about why SICP matters to computer scientists and computer professionals. It is clear that we decided to take up SICP based on the impressions of many stalwarts of the field. But it is one thing to read about an elixir and another to drink it. We have now decided to taste the elixir.

This document describes why we decided to take up SICP, how we made the allied decisions like the choice of the programming environment, and the notes and exercises during the entire course. The course is expected to last about 6 months (Nov 2022 - May 2023).

## Programming Environment

Lisp is great. One should get a taste of programming in Lisp. But neither Kedar nor Apoorv have gotten it. JavaScript has some “good parts.” The good parts of JavaScript are good parts of any programming language. Some sensible folks like Martin Henz and Tobias Wrigstad developed a [JavaScript Edition of SICP](#). They also provided [open access](#) to it! Samuel Fang developed the [Interactive JS interface](#) to SICP.

As a practical choice, I (Kedar) opted for JS. We'll read SICP JS and do all the exercises. From time to time we'll consult the [comparison edition](#) (that describes differences between the JS edition and the original). It is true that learning the Scheme dialect of Lisp opens up new vistas and school is perhaps the best place to learn Scheme. Still, using a subset of JavaScript to implement ideas in SICP feels like not such a bad decision because

- 1) JS is widely used everywhere (on the browser and server).
- 2) Good parts of JS are good. They make JS feel at least like a poor man's Scheme.
- 3) Learning Scheme is not a distraction, but it is another tool to learn and it may be better to defer doing that. However, I do think that during the undergrad years Apoorv should learn to think and code in Scheme (and get over the *parenthesitis*: initial mental inflammation due to an overuse of parentheses ;-)).

The programming language used by SICPJS is not one language. It is a progressively bigger language named [Source](#): at each “step” (or revision) it is a “sublanguage” of the next step. The final step and each previous step is, by definition, a sublanguage of the “full” JS:  
 $Source1 \subset Source2 \subset Source3 \subset Source4 \subset JS$ . They have also defined variants of each step. Looking at their [source code](#), it is clear that one can learn how to implement languages. I highly recommend taking a look at [it](#). The team at Source Academy has done [quite some work](#) on this!

# Dated (Counter-chronological) Notes and HW

23 February 2024

## [Exercise 1.40](#)

Declare a function, `cubic`, that can be used together with the `newtons_method` function in expressions of the form `newtons_method(cubic(a, b, c), 1)` to approximate zeros of the cubic  $x^3 + ax^2 + bx + c$ .

### **Solution.**

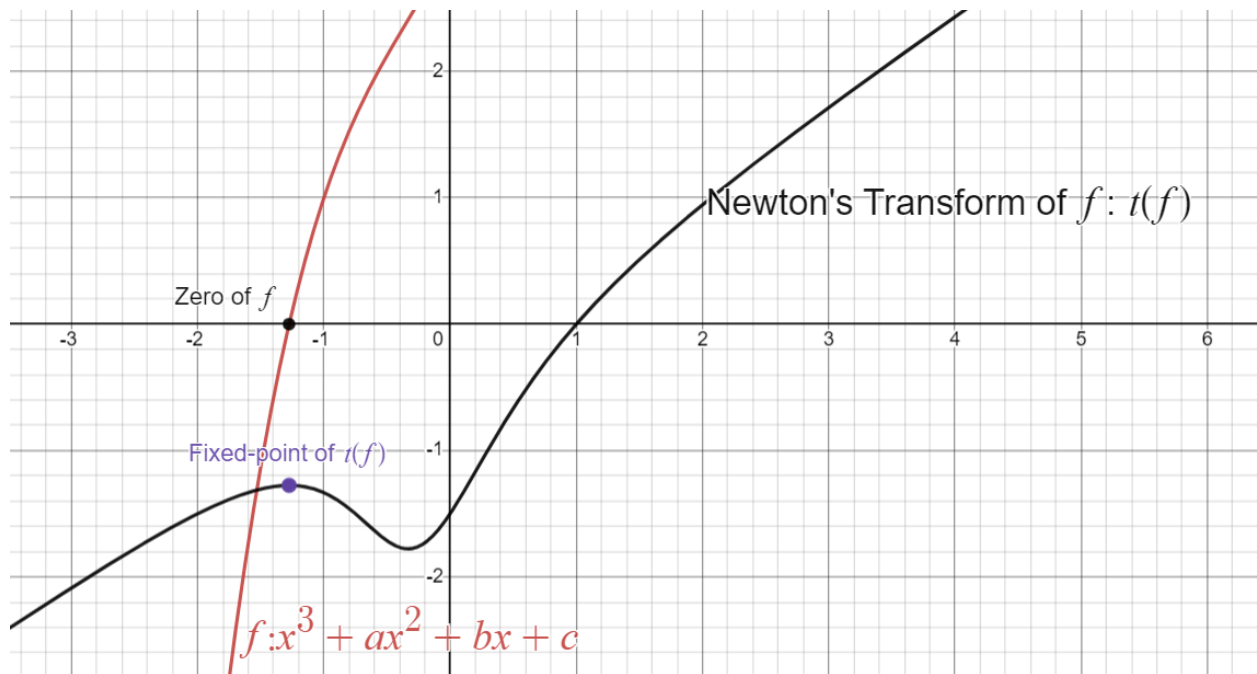
A polynomial function may not have a *real* zero. We need to address that problem, but let's first find out how to approximate a zero of a cubic that has a real zero. Newton's method is a way to do that: A zero of a function  $f$  is the fixed-point of its Newton's transform.

We want to define a function `cubic(a, b, c)` such that, for instance, `newtons_method(cubic(1, 2, 3), 1)`, returns an approximation of a zero of the function  $x^3 + x^2 + 2x + 3$ .

```
// returns a function that represents the cubic  $x^3 + ax^2 + bx + c$ 
function cubic(a, b, c) {
    return x => cube(x) + a * square(x) + b * x + c;
}

// returns a real number that is square of the given number, x
function square(x) {
    return x * x;
}
```

Desmos helps us plot this.



An application of `newtons_method`: `newtons_method(cubic(1, 2, 3), 1)` returns `-1.2756823846964038`, which is indeed an approximation of the zero of the given function.

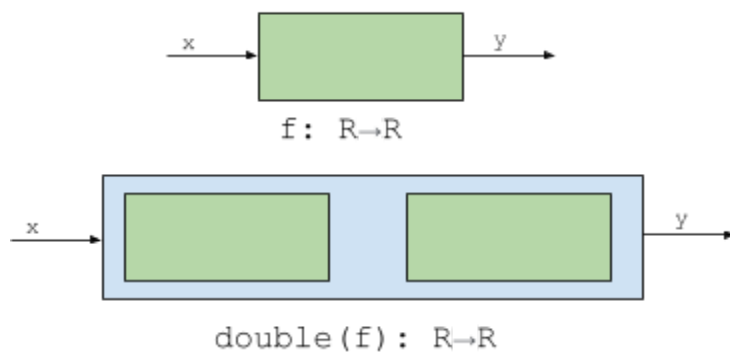
### Exercise 1.41

Declare a function `double` that takes a function of one argument as argument and returns a function that applies the original function twice. For example, if `inc` is a function that adds 1 to its argument, then `double(inc)` should be a function that adds 2. What value is returned by `double(double(double))(inc)(5)`?

#### Solution.

This is fairly straightforward.

```
function double(f) {
  return x => f(f(x));
}
```



That is how the functions, `f`, `double(f)`, work; both have their domain and codomain as the set of numbers (reals). Note that `double(f)` is the name of a function that takes a number and returns a number, e.g. `double(inc)(5)` applies `inc` to 5 first and then again to the number that (the previous) application returns. Thus, `double(inc)(5)` returns 7.

The function `double`, on the other hand, is the name of a (higher-order) function that takes a function and returns another function. Thus, `double(6)` has no meaning because 6 is not a function.

Wrapping your head around function declarations and function calls can be really tricky (albeit intellectually stimulating) in functional programming. The way it builds abstractions is pretty impressive and powerful, but, although the syntax (or the source code *line noise*) is thankfully minimal, the semantic load is rather high.

Let's analyze how the interpreter interprets `double(inc)`:

- 1) `double` is a function which means it can be called with `()`, the function call operator.
- 2) `double` applies to a single argument which is itself a function. `inc` is the name of a function; it qualifies as an argument to `double`.
- 3) `double(inc)` is a valid function application (or function call). The interpreter immediately calls the function. The result of this function application is another (hitherto unnamed) function. To be useful, it must be called immediately. We can make up a name for this function, say, `inc_doubler` although it is not necessary.
- 4) `double(inc)(5)` then can be renamed to `inc_doubler(5)` which applies 5 to `inc` that results in 6 and then again applies `inc` to 6 that results in 7 which it returns.

And that's it.

So far so good. The next challenge is to decipher `double(double(double))(inc)(5)`.

Before that, let's try `double(double)(inc)(5)`.

Will the *substitution model* help? Let's try. First, let's try to **expand** the definition of `double`. It's like expanding the function from its lambda-expression form:

```
function double(f) {  
  function d(x) {  
    const o1 = f(x);  
    return f(o1);  
  }  
  return d;  
}
```

Calling this function will return the inner function `d`.

So, what does the function call `double(double)` result in? **Right, it's still the inner function `d`.** But certain [binding of names](#) occurs because of that function call. The name `f`, which is local to `double`, is bound to the value `double`.

~~Let's give `double(double)` a name, say `d1`. Then, `double(double)(inc)(5)` will result in `d1(inc)(5)`.~~

Clearly, `double(double)` returns the inner function, `d`.

I think I lack a theoretical tool to analyze recursive functions properly. Till then, I guess I need to continue with the primitive methods I know.

**The question we need to answer is “How many calls to the function `inc` are made?” Or, thinking inductively, “if `n` successive appearances to `double` (i.e. `double(double(double))...`) in a source line result in `c` calls to `inc`, then `n+1` th appearance will result in how many calls to `inc`?”**

Perhaps tree-recursion will help.

[Distraction or not, I am taking a few days to learn the basics of  \$\lambda\$ c \(Lambda Calculus\).](#)

17 February 2024

## Abstractions and First-class Functions

Let's recap what we have done so far to find square-roots of real numbers.

First, we devised a general-purpose mechanism (we call it **abstraction**) to try to iteratively find a fixed-point (a real number) of a given function starting with a random guess. This abstraction was realized as a function, `fixed_point`. A caveat is that `fixed_point` may fail to notice that the given function has no fixed-point.

We tried to apply `fixed_point` to a function  $x \Rightarrow a/x$  (expressed here as a lambda expression) and some guess. That is when we identified a practical issue with the `fixed_point` function wherein a function `f` passed to it shows an oscillatory behavior: `f(a)` returns `b`, `f(b)` returns `a`, and none of `a` or `b` is close to `f`'s fixed-point. Therefore, applying `fixed_point` to `f` results in indefinite oscillation. Nothing is inherent in the `fixed_point` procedure to address this situation. Still, without changing the implementation of `fixed_point`, we remedied the situation by using another abstraction, average damping (function `average_damping`), applied to the given function:  $x \Rightarrow \text{average}(f(x), x)$ .

We kept the `fixed_point` abstraction untouched, introduced a transform of the given function, and used that transform to calculate the square-root of a real number `a`:

```
fixed_point(average_damping(x => a/x), 1.0).
```

This gave birth to a generally useful separate abstraction, `average_damping`. The fixed-point of `f` and its *average-damped transform* are the same. We thought that, to find `f`'s fixed-point, we might as well find `average_damping(f)`'s fixed-point. In doing so, we addressed the oscillatory behavior of `fixed_point` that hindered its convergence (how quickly it reaches the fixed-point).

Then we discovered Newton's Method (function `newton_method`) which said that if for a function `f(x)`, that has a zero (value of `x` for which `f(x)` is 0), `g` is an approximation of its zero, then  $g - f(g)/f'(g)$  is a better approximation. We called  $g - f(g)/f'(g)$  Newton's Transform (function `newton_transform`) of `f`. **It turned out that the fixed-point of Newton's Transform of `f` coincided with a zero of `f`.** Thus, we discovered another general purpose abstraction. We used that to find the square-root of a real number `a`:

```
fixed_point(newton_transform(x => square(x) - a)).
```

Thus, in both these methods, we applied `fixed_point` *not* directly to a function but to a certain transform of it.

Is there another abstraction lurking here? Can we express this process in order for it to be more generally useful?

This question often comes when one is learning to define *abstractions*, the heart of computer programming. Only with enough practice can one get better at creating generally useful abstractions.

In the present case, is abstracting out transform to another function generally useful? When finding square-root by the first method, we commit to average damping, whereas when doing so by the second method, we commit to Newton's transform. Can we abstract that out in the form of a new parameter? Perhaps we can.

Consider a new function: `fixed_point_of_transform(f, transform, guess)`. This function dissociates the transform from fixed-point iteration and passes on the choice of a transform to the code that *applies* or *calls* this function. This is also known as *behavior parameterization*. This function is pretty straightforward:

```
function fixed_point_of_transform(f, transform, g) {  
    return fixed_point(transform(f), g);  
}
```

Now we have two ways to find the square-root of a real number by choosing appropriate transforms and functions:

```
function square_root_1(a) {  
    return fixed_point_of_transform(x => a/x, average_damping, 1);  
}  
function square_root_2(a) {  
    return fixed_point_of_transform(x => square(x) - a,  
    newton_transform, 1);  
}
```

Does this enhance your power to express yourself through programs? Does this ease of expressiveness matter to you?

Answers to such questions may be subjective. But one thing is clear: Providing fundamental abstractions is like providing fundamental tools of trade. It is up to the craftsmen to utilize these tools to their advantage. A sort of hierarchy of programmers emerges here. Perhaps the first in this hierarchy is the hardware designer who provides the basic instruction set of a computer and things happen in the physical layer – the semiconductor. The amount of flexibility that they can offer to others is limited because of the complexity involved at the “low level.”

A higher-level programming language designer is perhaps the next level. They provide a hopefully minimal set of rules and programming elements to other programmers who solve real problems of the end users. A function is a fundamental programming element. Every programming element has certain restrictions. The elements that have fewest restrictions are called the **first-class citizens** (of that programming language). Features of every first-class citizen include:

- 1) Referral using *names*,
- 2) Use as *arguments* of functions,
- 3) Use as *values returned* from functions,
- 4) Use in data structures.

Making functions first-class citizens of a language is a conscious decision that a programming language designer makes. When functions are first-class citizens, they demonstrate all the above capabilities (of course that comes at a cost, but the cost is justified). The computer scientist Christopher Strachey is often credited with making functions first-class citizens of programming languages.

21 January 2024

## Newton's Method

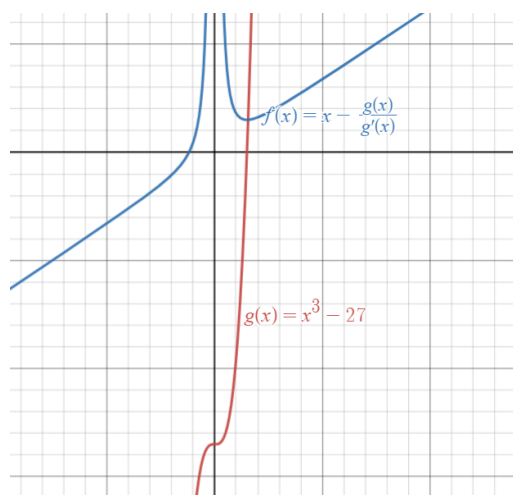
Being stuck is not fun. I was stuck on the [inverse problem](#) for quite some time: **To which function can the fixed-point iteration be applied to get the solution of the equation  $g(x) = 0$  (provided  $g(x)$  is differentiable)?**

And this section provides a readymade answer:

The solution of  $g(x) = 0$  is the fixed-point of the function  $f - f(x) = x - \frac{g(x)}{g'(x)}$ , where

$g'(x)$  is the derivative (function) of  $g(x)$ :  $g'(x) = \lim_{h \rightarrow 0} \frac{g(x+h) - g(x)}{h}$ . (The limit should be

defined because  $g$  is differentiable.)



And the classical definition of [Newton's method](#) is about approximating a solution of  $g(x) = 0$ . It says that **if  $g$  satisfies certain conditions and  $x_0$  is an approximation for the solution of  $g(x) = 0$ , then  $x_0 - \frac{g(x_0)}{g'(x_0)}$  is a better approximation.**

'Better Approximation' is a qualitative term. We'll see what *better* means.

The key idea in Newton's method is  $g'(x)$  which is itself a function. We call it the derivative (function) of  $g(x)$ . Since the derivative of a given differentiable function is a function, after making sure it exists, we

can calculate its value at a given value of  $x$ .

What the [above](#) says can be illustrated by an example. Consider the function

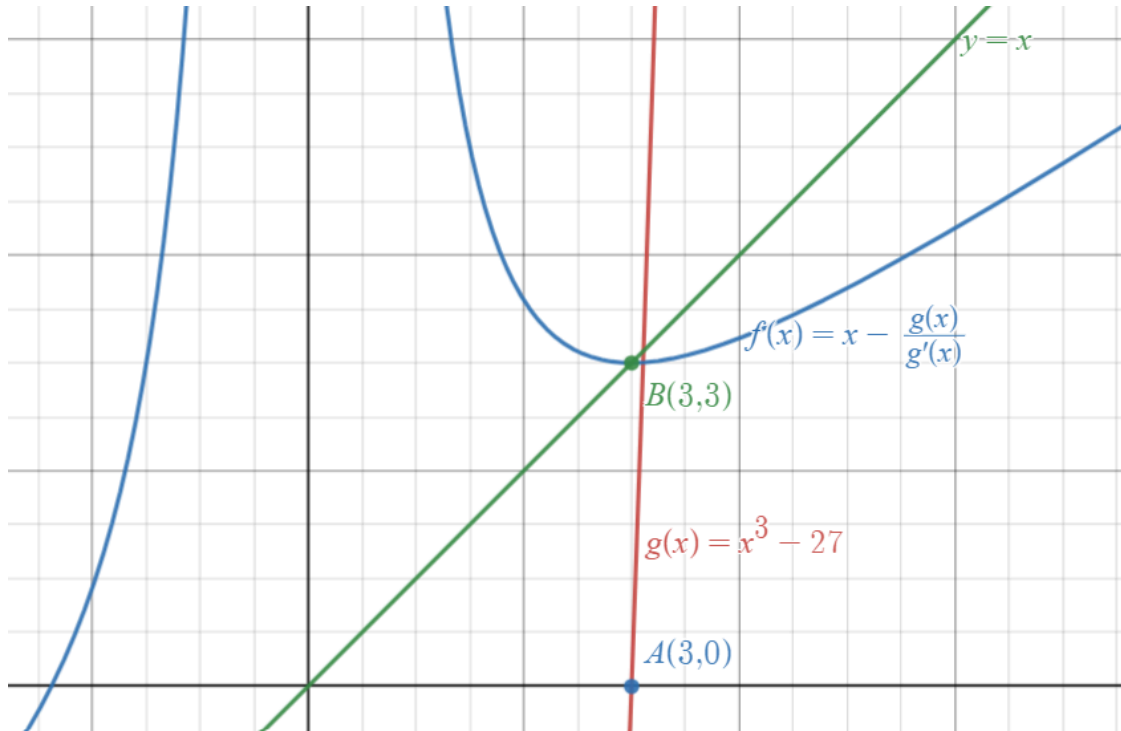
$g(x) = x^3 - 27$ , whose root we want to find (assuming a *real* root exists). Then,

$f(x) = x - \frac{x^3 - 27}{3x^2}$ . Both the functions are plotted in the adjacent figure. **The claim is**

**that the solution of the polynomial (in red) is the fixed-point of the function in blue.**

This is illustrated in the figure below.





The point A depicts the solution of  $g(x)$ . The point B depicts the fixed-point of  $f(x)$ . **The claim is that the X-coordinates of A and B are the same.**

Come back to a proof of this.

Once we accept this, however, it is a small matter of programming it clearly.

Derivative function can be clearly expressed as a function that is returned by another function (along with a declaration for a sufficiently small positive real number,  $dx$ ):

```
const dx = 1e-5;
function deriv(g) {
    return x => (g(x + dx) - g(x)) / dx;
}
```

This little function, `deriv`, like that little function `average_damping`, provides enormous expressive power: when called on a point  $(x)$ , it finds the value of the derivative of a differentiable function at that point:

```
function cube(x) {
    return x * x * x;
}
```

Note that `deriv(cube)` is just a function that is waiting to be called.

```
deriv(cube)(5);
```

// returns 75.00014999664018 which is the value of the derivative of the `cube(x)` function at `x = 5`

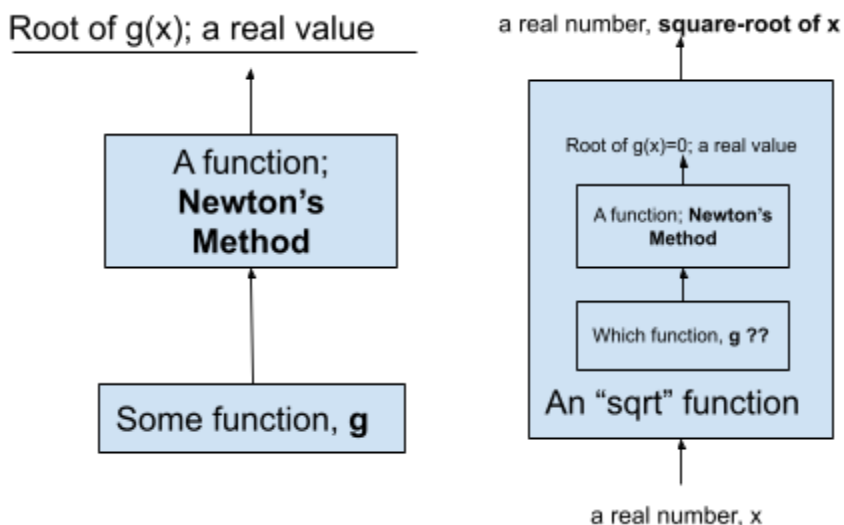
Now, Newton's method can be more effortlessly transformed into a fixed-point method:

```
function newton_transform(g) {  
    return x => x - g(x) / deriv(g)(x);  
}  
// Finds a root of g  
function newtons_method(g, guess) {  
    return fixed_point(newton_transform(g), guess);  
}
```

In a straightforward manner, Newton's method, with a function and starting with a reasonable guess, becomes the fixed-point iteration applied to Newton's transform of that function and starting with that guess. **Newton's method then finds a root (or zero) of the given function.**

I think that the whole endeavor in functional programming is about being able to express oneself simply.

How can we use the latest realization that Newton's method finds a root of a given function to find the square root of a real number? This question seems to recur. Has my understanding of the machinations improved? Here is the puzzle represented pictorially:



**Using** the `newtons_method` function (knowing what it does), can we devise another function, say `sqrt`, that when applied to a real number returns its square-root?

As I was waking up on the morning of 08 Feb 2024, some things became clearer (and Knuth's *Literate Programming* dawned upon me once again as I ran into a [computer algebra system called Axiom](#)):

1) The Wikipedia page on [root-finding methods](#) says:

[Solving an equation](#)  $f(x) = g(x)$  is the same as finding the roots (zeros) of the function  $h(x) = f(x) - g(x)$ . Thus root-finding algorithms allow solving any equation defined by continuous functions.

2) This leads to the idea that square-root of a positive real number  $a$  is same as the root (zero) of the function  $f(x) = x^2 - a$ . Starting from some value of  $x$ , we want to move to that value of  $x$  whose square is *almost* the same as  $a$  (if not the same as  $a$ ).

On 13 February, back in Pune, I decided to write it down again with the help of a developing picture.

Consider that I want to find the square root of a positive real number  $a$  represented on a number line by the point A. On the coordinate plane, coordinates of A are  $(a, 0)$ . Let the square root of A be represented by the point T. T's coordinates are  $(\sqrt{a}, 0)$ . In our figure (Figure 0213.1), T is to the left of A, but that does not matter. We only symbolically know the coordinates of T. Our task is to locate T using a general procedure.

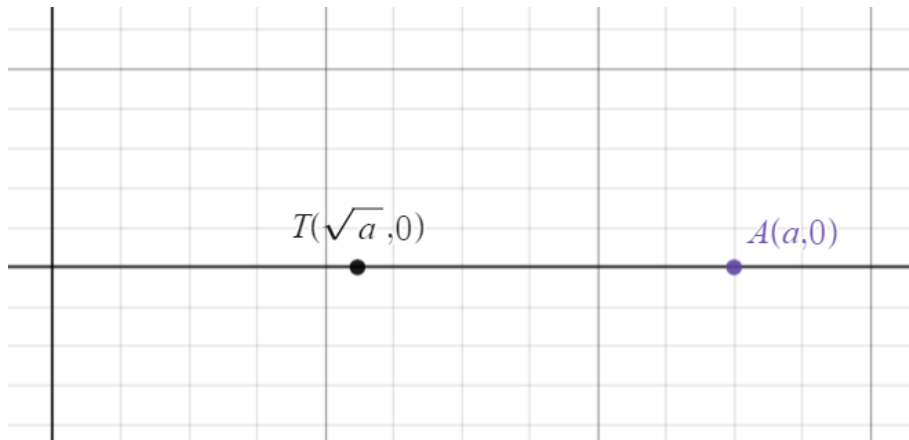


Figure 0213.1: Locate T, given A.

The key idea we need to realize is that  $T(\sqrt{a}, 0)$  is a **zero of the polynomial**  $x^2 - a$ . See Figure 0213.2. **Without this realization, further discussion is not helpful.**

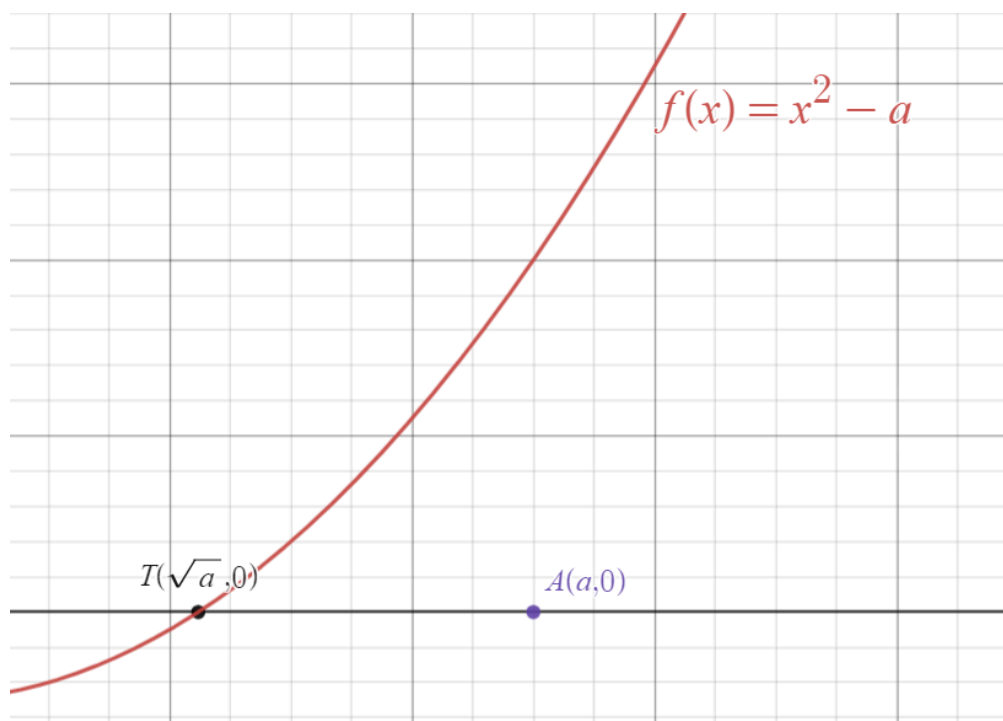


Figure 0213.2 Square root as a zero of a polynomial.

Now, given this polynomial, we want to locate, even approximately, the point T using some general procedure. Some people call that procedure *linear approximation*.

We make a guess; start with some point  $G_0$  on the X-axis. It does not matter where  $G_0$  is. We examine if  $G_0$  is **sufficiently close** to T. The test we use to determine if they T and  $G_0$  are close enough is straightforward – verify if the square of  $G_0$ 's X-coordinate is close to  $a$ . If yes, we are done. We found  $\sqrt{a}$  at  $G_0$ . But chances are that T and  $G_0$  are far apart. See Figure 0213.3 where we illustrate our start with  $G_0$  as  $(\frac{a}{2}, 0)$ .

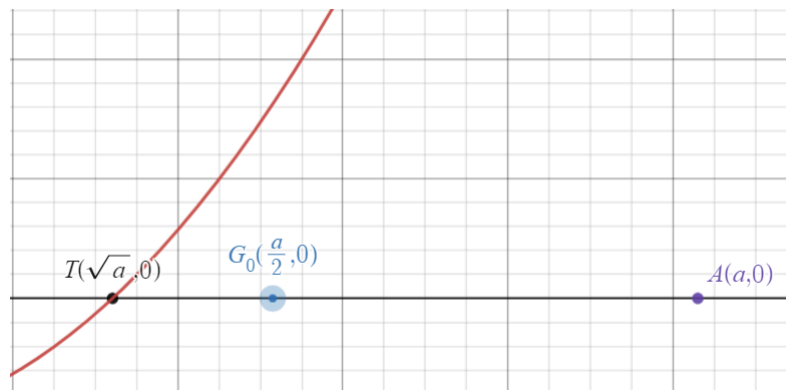


Figure 0213.3: Start with a random guess  $G_0$ , not close enough to T.

Now we use what Newton's method proposes. It asks that we find the point  $P(g_0, f(g_0))$

on the curve that represents our polynomial:  $f(x) = x^2 - a$ ,

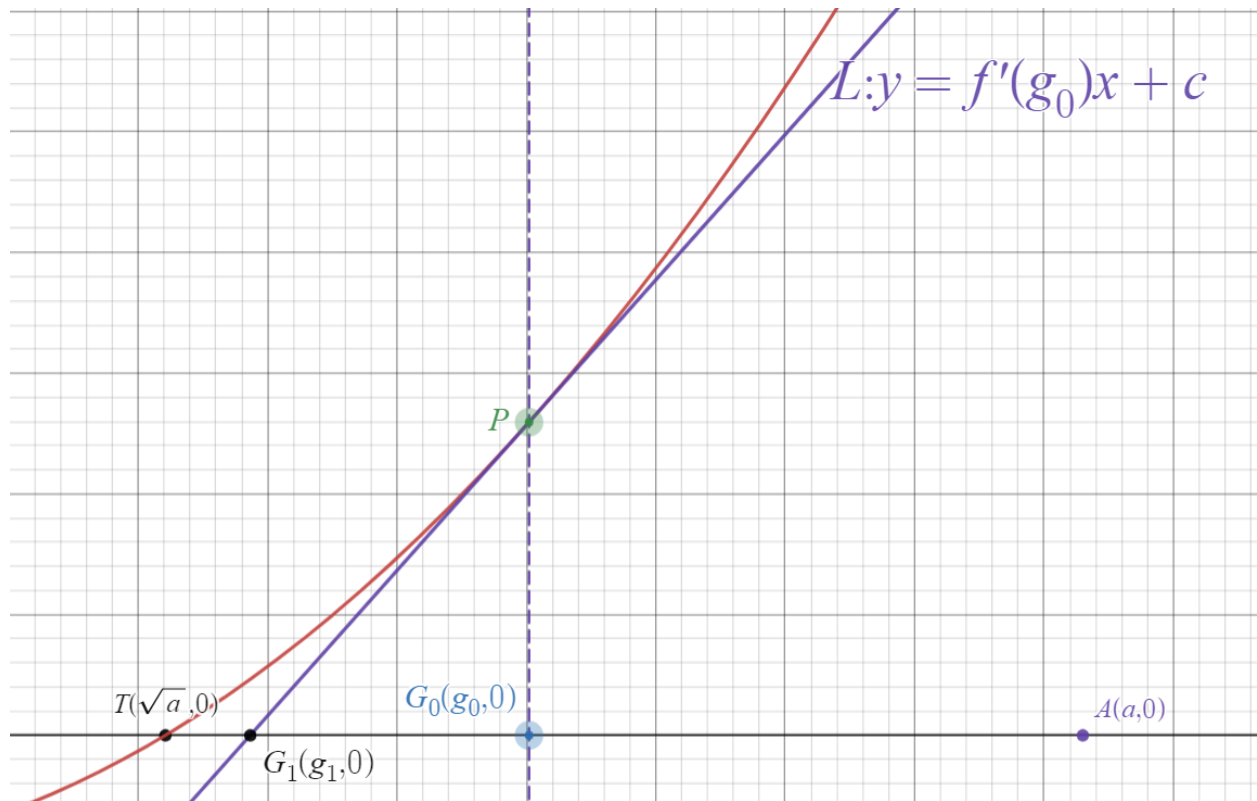


Figure 0213.4 Newton's Method, first approximation,  $G_1$ .

and draw a tangent  $L$  to it at  $L$ . Let  $L$  intersect X-axis in  $G_1$  (this will always happen as long as  $L$  is not parallel to X-axis, or, in other words,  $f'(g_0) \neq 0$ ). Then  $g_1$ , the X-coordinate of  $G_1$ , is a better approximation of  $\sqrt{a}$  than  $g_0$ . We use the derivative in finding  $L$ : The slope of  $L$  equals  $f'(g_0)$ .

We now repeat the procedure:  $G_1$  is the new  $G_0$ .

This is the theory of Newton's method. How do we weave our functional programming story around it?

We have two basic abstractions:

- 1) The function `fixed_point`: It finds the fixed-point (a real number) of a given function (if it has one) starting from a random guess. Unfortunately, it does not know a priori if a given function does not have a fixed-point.

- 2) The function derivative: It finds the value of a derivative (a real number) of a given function at a particular value. It assumes that the given function is differentiable.

We need to use them to our advantage: to find the square-root of a real number, say,  $a$ . Given  $a$  (represented by the [point A](#)), we need to find  $\sqrt{a}$  (i.e. reach the [point T](#)). The above discussion tells us that in order to find a zero of  $f(x)$ , we need to find the fixed-point of Newton's transform of  $f(x)$ . For the square-root of  $a$ , we just need to do that to the function  $x^2 - a$ :

```
function sqrt(a) {  
  return newtons_method(x => square(x) - a, 1);  
}
```

Here's a more complete listing:

```
function fixed_point(f, guess) {  
  const e = 1e-5;  
  function try_with(curr) {  
    const next = f(curr);  
    return math_abs(curr - next) <= e  
      ? curr  
      : try_with(next);  
  }  
  return try_with(guess);  
}  
  
function square(x) {  
  return x * x;  
}  
  
const h = 1e-6;  
function deriv(g) {  
  return x => (g(x+h) - g(x)) / h;  
}  
  
function newton_transform(g) {  
  return x => x - g(x)/deriv(g)(x);  
}  
  
function newtons_method(g, guess) {  
  return fixed_point(newton_transform(g), guess);  
}
```

```
function sqrt(x) {
  return newtons_method(y => square(y) - x, 1);
}
sqrt(89); // 9.433981132085753
```

19 January 2024

I finally finished the exercises on 1.3 and am ready to move on. I think I have learned enough about recursive and iterative processes and applying functions to other functions (or, in other words, passing functions as function arguments to other functions). The simple ability to pass functions to other functions provides expressive power to your programs.

One thing to bear in mind is that although functions (like ordinary numbers) can be passed to other functions, only functions (unlike ordinary numbers) can be applied to arguments. We capture this speciality of functions in: ***Functions can be called.***

### 1.3.4 Functions as Return Values

Just like we can pass functions to other functions, we should be able to return them from other functions as well; and we can!

The book aptly describes what we have done and what we can do with functions as return values:

We can illustrate this idea by looking again at the fixed-point example described at the end of section [1.3.3](#). **We formulated a new version of the square-root function as a fixed-point search, starting with the observation that  $\sqrt{x}$  is a fixed-point of the function  $y \rightarrow \frac{x}{y}$ .** Then we used average damping to make the approximations converge. **Average damping is a useful general technique in itself. Namely, given a function  $f$ , we consider the function whose value at  $x$  is equal to the average of  $x$  and  $f(x)$ .**

Consider the following function:

```
function average_damp(f) {
  return x => average (x + f(x));
}
```

It accepts a function and **returns a function which, when invoked, does the job of average damping**. What this function returns must be applied to some other arguments:

```
average_damp(square)(10); // => 55 - in-place application
```

Let's write down clearly what various functions are doing:

- 1) `sqrt(x)`: accepts a number and returns a number that is its square-root.
- 2) `fixed_point(f, g)`: accepts a function `f` and an initial guess and returns the number that is `f`'s fixed-point.
- 3) `average_damp(f)`: accepts a function `f` and returns another function that when applied to a number, say, `x`, returns  $\frac{1}{2}(x + f(x))$ .

Recall how we found the square-root of a number using the fixed-point iteration and average damping for faster convergence:

```
function sqrt(x) { // (1)
  return fixed_point(y => average(y, x / y), 1);
}
```

How can this procedure be expressed differently now that we know that functions can be returned as values from other functions (like the function `average_damp`)?

It turns out that (we added `*` just for identification, it serves no other purpose)

```
* function sqrt(x) { // (2)
  return fixed_point(average_damp(y => x / y), 1);
}
```

is equivalent to the above function.

Does [this version of `sqrt\(x\)`](#) (2) have more expressive power than [that version of `sqrt\(x\)`](#) (1)? Is this a question of aesthetics?

It is perhaps difficult to choose. But the second way seems to more simply translate into code the following: you get the square-root of a real number when the fixed-point iteration is applied to the average-damped version of the function: `y => x / y` and a suitable guess.

The average-damped version of the function `y => x/y` is simply **returned** by **`average_damp(y => x / y)`** when `fixed_point` is called (remember the [applicative-order](#) evaluation?). The function `y => average(y, x/y)` is also an average-damped version of the function `y => x/y`. What's the difference then?

One difference is that the version (2) abstracts the process of averaging out of the function to which fixed-point iteration is applied. In version (1), that function carries out averaging itself.



Is that such a big difference? I think it is a matter of perspective and experience. If average-damping is deemed a general facility that promotes code reuse then it is more applicable.

At the same time, arguably, the overall structure has become slightly more complex. Is that complexity justified? How much abstraction is too much? These questions are difficult to answer in isolation. Some contextual knowledge of the overall module or library that delivers a particular functionality in software is required to satisfactorily answer such questions. The book says that good experienced programmers develop the ability to write such lucid abstractions that are more generally applicable.

Here, having an abstraction of two kinds:

- 1) An ability to do average-damping, and
- 2) An ability to return functions from other functions

seems to be of some very good general use.

Let's now demonstrate the use of this abstraction to take the cube-root of a real number:

```
function cube_root(x) {  
    return fixed_point(average_damp(y => x/square(y), 1);  
}
```

Or, equivalently, using the older construct:

```
function cube_root(x) {  
    return fixed_point(y => average(y, x/square(y)), 1);  
}
```

Another subtle point is that of syntax. Source1's syntactic sugar just enhances the expressive power of abstractions (which relates to semantics) . Consider the following version of the average\_damp function:

```
function average_damp(f) {  
    function g(x) {  
        return average(x, f(x));  
    }  
    return g;  
}
```

This is equivalent to

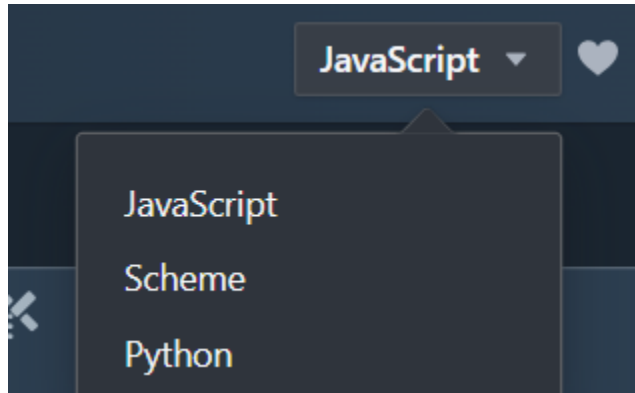
```
function average_damp(f) {
```

```
    return x => average(x, f(x));
}
```

but rather less compact.

14 January 2024

I believe I am keeping the SICP flame burning. Writing functional programs to solve problems keeps me interested. I guess I should increase the pace. I am doing many things anyway.



Source Academy guys are awesome. I just noticed that they added support for Scheme and Python on the playground!

### [Exercise 1.39](#)

A continued fraction representation of the tangent function was published in 1770 by the German mathematician J.H. Lambert:

$$\tan x = \frac{x}{1 - \frac{x^2}{3 - \frac{x^2}{5 - \frac{x^2}{\dots}}}}$$

**Solution.**

```
// We reuse the continued fraction's iterative process
function cont_frac(n, d, k) {
  function sum(curr, acc) {
    const partial = n(curr) / (acc + d(curr));
    return (curr <= 1)
      ? partial
      : sum(curr - 1, partial);
  }
}
```

```

    return sum(k, 0);
}

// J. H. Lambert's approximation of the tan function as a cont_frac
function tan_cf(x, k) {
    return cont_frac(i => i <= 1 ? x : -x * x, i => 2 * i - 1, k);
}
const theta = math_PI / 10;
const k = 100;
math_tan(theta) - tan_cf(theta, k); // difference must be very small
// prints -5.551115123125783e-17 on my computer
(Again, I can not stop wondering how Lambert came up with this formulation!)

```

### Exercise 1.38

In 1737, the Swiss mathematician Leonhard Euler published a memoir, *De Fractionibus Continuis*, which included a continued fraction expansion for  $e - 2$ , where  $e$  is the base of the natural logarithms. In this fraction, the  $N_i$ 's are all 1 and  $D_i$ 's are successively 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, .... Write a program that uses your `cont_frac` function from exercise [1.37](#) to approximate  $e$ , based on Euler's expansion.

### **Solution.**

This is an exercise in devising and applying the functions `n` and `d` (required by the `cont_frac` function) correctly.

```

// We reuse the continued fraction's iterative process
function cont_frac(n, d, k) {
    function sum(curr, acc) {
        const partial = n(curr) / (acc + d(curr));
        return (curr === 1)
            ? partial
            : sum(curr - 1, partial);
    }
    return sum(k, 0);
}

// For e, n_i's are all 1, d_i's are drawn from this curious sequence:
// 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, ...
// Expressing that as a function is perhaps not that difficult, but
// I keep wondering how Euler came up with that!

```

```
// function dfun(i) {
//     const d = i % 3 < 2
//         ? 1
//         : 2 * (math_ceil(i/3));

//     display(d);
//     return d;
// }
cont_frac(i => 1, i => i % 3 < 2 ? 1 : 2 * (math_ceil(i/3)), 100); //
prints 0.7182818284590453 ~ e-2
```

31 October 2023

Exercises on `fixed_point`.

### [Exercise 1.35](#)

Show that the golden ratio  $\varphi$  (section [1.2.2](#)) is a fixed point of the transformation  $x \rightarrow 1 + \frac{1}{x}$  and use this fact to compute  $\varphi$  by means of the `fixed_point` function.

**Solution.**

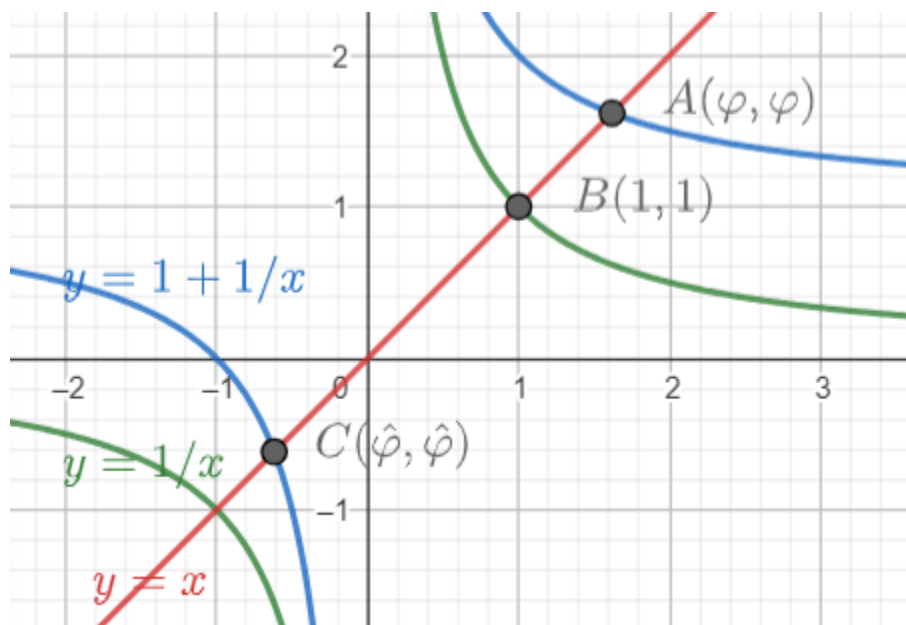


Figure 1.35.1 – Graphs showing fixed points of certain transformations

Showing that some value is a fixed point of a function (or transformation)  $y = f(x)$  is to solve the system of equations:

$$y = f(x) \text{ ----- (1.35.1)}$$

$$y = x \text{ ----- (1.35.2)}$$

Or, in other words, finding the value of  $x$  that satisfies the equation  $x = f(x)$  ----- (1.35.3).

Given the transformation, we get:

$$1 + \frac{1}{x} = x$$

$\therefore x^2 = x + 1 \Rightarrow x^2 - x - 1 = 0 \Rightarrow x = \frac{1 \pm \sqrt{5}}{2}$  which are the two values,  $\varphi$ , and  $\hat{\varphi}$  (the golden ratio and its conjugate).

Applying `fixed_point` to the lambda expression  $x \Rightarrow 1 + 1/x$  yields  $\varphi$ .

### [Exercise 1.36](#)

Modify `fixed_point` so that it prints the sequence of approximations it generates, using the primitive function `display` shown in [exercise 1.22](#). Then find a solution to  $x^x = 1000$  by finding the fixed-point of  $x \mapsto \frac{\log(1000)}{\log(x)}$ . (Use the primitive function `math_log` which computes the natural logarithms.)

Compare the number of steps this takes with and without average damping. (Note that you cannot start with a guess of 1, as that would cause division by 0 (since  $\log(1) = 0$ )).

15 November 2023

#### **Solution.**

The first part is easy. Displaying the sequence of approximations is just calling the `display` function with appropriate arguments.

```
const tolerance = 0.0001; // some small value
function fixed_point(f, guess) {
  function close_enough(a, b) {
    return math_abs(a - b) < tolerance;
  }
  function try_with(guess) {
    const next = f(guess);
    display("guess ", i, " = "
    return close_enough(guess, next)
      ? next // one may return guess
      : try_with(next);
  }
}
```

```

    return try_with(guess);
}

```

The second part is rather tricky. It again brings back the questions about the inverse problem: How do I relate the solution of an equation for  $f(x) = 0$  by finding the fixed-point of a function  $g(x)$ ? Finding the fixed-point of which function will yield the solution of a given equation?

I asked this question to Apoorv (visiting home, December 2023), but he too couldn't convincingly say something.

The exercise gives away the answer to that crucial question by simply stating that

Solution of the equation,  $x^x = 1000$ , is the fixed-point of the transformation  
 (function):  $x \rightarrow \frac{\log(1000)}{\log(x)}$ .

One way to think about it is to express the given equation (whose solutions we seek) in a way that, somewhat atypically, makes the indeterminate ( $x$ ) appear on both sides of the equation.

The equation  $x^x = 1000$  is equivalent to (by taking logarithms on both sides – that should be allowed since the log function is defined and monotonically increasing on positive real numbers and we know that  $x$  is a positive real number between 4 and 5 (

$1^1 = 1, 2^2 = 4, 3^3 = 27, 4^4 = 256, 5^5 = 3125$ )):

$x \log(x) = \log(1000) \rightarrow x = \frac{\log(1000)}{\log(x)}$ . Once we are able to express the given equation thus, maybe the fixed-point iteration takes care of finding the approximate solution iteratively. We then use average damping to improve the rate of convergence.

Here is the code that adds some reporting to the fixed-point iteration:

```

const tolerance = 0.0001; // some small value
function fixed_point(f, guess, n_iter) {
  function close_enough(a, b) {
    return math_abs(a - b) < tolerance;
  }
  function try_with(guess, n_iter) {
    const next = f(guess);
    display(guess, ": current");
    display(next, ": next");
    display(n_iter, ": n_iter");
  }
}

```

```

        return close_enough(guess, next)
            ? next // one may return guess
            : try_with(next, n_iter + 1);
    }
    return try_with(guess, n_iter);
}

function average(x, y) {
    return (x + y) / 2;
}

```

This application:

fixed\_point(x => math\_log(1000)/math\_log(x), 2, 1);  
 converges to 4.555563237292884 in 29 iterations, whereas this one, with average damping,  
 converges to 4.555465521473675 in just 8 iterations:

```
fixed_point(x => average(x, math_log(1000)/math_log(x)), 2, 1);
```

### [Exercise 1.37](#)

An *infinite continued fraction* is an expression of the form

$$f = \frac{N_1}{D_1 + \frac{N_2}{D_2 + \frac{N_3}{D_3 + \dots}}}$$

As an example, one can show that the infinite continued fraction with  $N_i = D_i = 1 \forall i$  produces  $\frac{1}{\phi}$  where  $\phi$  is [the golden ratio](#).

#### 1.37.1

Suppose that  $n$  and  $d$  are functions of one argument (the term index  $i$ ) that return  $N_i$ ,  $D_i$  of a continued fraction respectively.

Declare a function `cont_frac` such that evaluating `cont_frac(n, d, k)` computes the value of the  $k^{th}$  finite continued fraction. Check your function by approximating  $\frac{1}{\phi}$  using

```
cont_frac(i => 1, i => 1, k)
```

for successive values of  $k$ . How large must you make  $k$  in order to get an approximation that is accurate to 4 decimal places?

### Solution.

Continued fractions are very interesting. Infinite continued fractions are even more so. I am going to study them in more detail. MAA has published two books, one by C. J. Olds and another by Andrew Simoson. But this problem seems fairly straightforward.

```
// Implement the finite continued fraction using an iterative process.
```

```
// n is a function that returns the numerator of the ith fraction
```

```
// d is a function that returns the denominator of the ith fraction
```

```
// k is the number of times to iterate
```

```
function cont_frac(n, d, k) {  
  function sum(curr, acc) {  
    const partial = n(curr) / (acc + d(curr));  
    return (curr === 1)  
      ? partial  
      : sum(curr - 1, partial);  
  }  
  return sum(k, 0);  
}
```

```
display(cont_frac(i => 1, i => 1, 10)); // prints 0.6179775280898876
```

```
const phi_mul_inv = 0.6180339887498948; //1/φ
```

```
const diff = 0.00000001;
```

```
// Find the number of iterations it takes for a given continued_fraction
```

```
// function to reach a given limit.
```

```
function find_conv_iters(cont_frac_calculator, nfun, dfun, k, limit) {  
  return math_abs(cont_frac_calculator(nfun, dfun, k) - limit) < diff  
    ? k  
    : find_conv_iters(cont_frac_calculator, nfun, dfun, k + 1, limit);  
}
```

```
display(find_conv_iters(cont_frac, i => 1, i => 1, 1, phi_mul_inv)); // prints  
19
```

It takes 19 invocations on my computer to converge to  $1/\phi$ . I have extracted the common behavior and created the above function, `find_conv_iters`. It works with different functions parametrized by `cont_frac_calculator` that calculate the (finite) continued fraction.



### 1.37.2

If your `cont_frac` function generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

#### Solution.

Our [procedure above](#) is an iterative one since there are no outstanding operations after the recursive calls return. For that to happen we calculate the last fraction first. If we implement a more literal order, we end up having a recursive process because we have an *addition* operation after each recursive call returns:

```
// Implement the finite continued fraction using a recursive process.
// n is a function that returns the numerator of the ith fraction
// d is a function that returns the denominator of the ith fraction
// k is the number of times to iterate
function cont_frac_rec(n, d, k) {
  function sum(curr) {
    return curr === k
      ? n(k)/d(k)
      : n(k)/(d(k) + sum(curr + 1));
  }
  return sum(1);
}
```

13 October 2023

I resumed reading/programming SICP JS because I wanted to better understand JavaScript promises. I have not yet gotten to them. But I'll perhaps get there soon.

For now, I am continuing with Fixed-point. The Babylonian method of finding square-root of a number by using the iterative fixed-point procedure seems to take time. I need to take some time to understand that better. See [below](#).

10 October 2023

Many things have changed since I last visited here. I must continue the SICP trek albeit alone. Apoorv is now at Michigan. I miss our study sessions a lot. We were really good buddies. But I am very happy for him. I hope he does bigger and better things at UM. Back to work now.

22 June 2023

### 1.3.3. Functions as General Methods

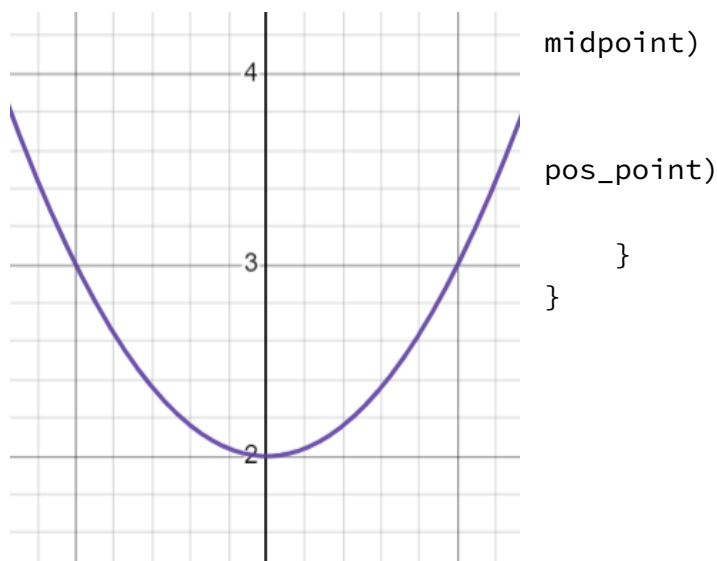
Consider a function like [riemann\\_sum](#) (simple integral) or [simpson](#) (Simpson's rule). It is applied to several arguments, one of which is a function. These functions represent general, well-defined procedures that can be interpreted as methods involving other (mathematical) functions. Integration can be thought of as a function that operates at a higher level than a function that, for example, squares a number. We often say a definite integral represents the area under a curve. Thus, given any curve (continuous), it finds the area of the shape enclosed between it and the X-axis.

Typically, functions we encounter in *Algebra* are like first-order functions, whereas many functions we encounter in *Analysis* are like higher-order functions. Some general procedures in Algebra can also be realized by the way of higher-order functions. Such higher-order functions are what are meant by "General Methods" here.

#### The half-interval method

Take, for example, the half-interval **method** of finding roots of an equation represented by a continuous (mathematical) function (the function can be algebraic, rational, exponential, trigonometric, or logarithmic). This method is general enough that it applies to any function as long as we know two numbers (however far apart) at which the function's values have opposite signs. This **method** is a standard iterative technique of searching a value in a collection of values that utilizes an enabling property. That property has to do with the orderliness (or monotonicity) of values in the given collection. Here is an implementation:

```
function search(f, neg_point, pos_point) {  
  const midpoint = average(neg_point, pos_point);  
  if (close_enough(neg_point, pos_point)) {  
    return midpoint;  
  } else {  
    const test_value = f(midpoint);  
    return positive(test_value)  
  }  
}
```



```
? search(f, neg_point,  
midpoint)  
: negative(test_value)  
? search(f, midpoint,  
pos_point)  
: midpoint;  
}  
}
```

Given two points however far apart where a continuous function has values of opposite signs, this iterative procedure is bound to reach the vicinity of the zero of that function because we maintain at every step an **invariant** that is preserved in the beginning. At every step, we solve a problem that has the same form (of a smaller size) as the original.

### Fixed points of functions

A clear definition of “mathematical” function which is another name for “transformation”, requires some background, but, roughly speaking, a function returns the same value every time it is given the same value. In other words, a function does not **output** two or more values for the same **input**.

We can imagine functions returning values that are the same as the values that they accept. If we think of functions as black-boxes, then we can imagine some input, say  $x_{FP}$ , that remains sort of unfazed by the functional transformation. Notationally,  $f(x_{FP}) = x_{FP}$ . This  $x$  which just falls through the function black-box (untransformed) is called its “fixed-point”.

We can plot mathematical functions on a coordinate plane. Some reflection will show that a function’s fixed-point is the same as the point of its intersection with the straight line  $y = x$ . Clearly, not every function has a fixed point because not every function’s plot intersects  $y = x$ . For example, the function  $y = x^2 + 2$  has no fixed-point (see the diagram).

How does one relate the idea of the **fixed-point of a function** with the idea of the **square root of a positive real number**?

This question troubled me a lot. I struggled to understand the connection between these two ideas.

First, finding the fixed point of a mathematical function,  $f(x)$ , mathematically means solving the equation  $f(x) = x$ . If we know the theory of equations, this can be solved on paper. For the function  $x^2$ , for instance, we can solve the equation:  $x^2 = x \Rightarrow x = \{0, 1\}$ . These two are the fixed-points of this function.

How about the fixed-point of  $x^2 + 2$ ? At the fixed-point of this function,  $x^2 + 2 = x$ , or  $x^2 - x + 2 = 0$ , and we get:  $x = \frac{1 \pm \sqrt{1-8}}{2} = \frac{1 \pm \sqrt{7}i}{2}$ . Thus, fixed-points of this function exist, but they are complex! (This is also an interesting observation – What is not possible in the real domain is possible in the imaginary domain! This is a sort of epiphany for me because now one can imagine that imaginary numbers can be roots of at least more polynomials – once we are willing to enhance our notion of a number to include square roots of -1, we can suddenly find roots of many, many (all?) polynomials.)

How do we use a computer program to do the same? One way is to repeatedly apply the function to a value starting with some convenient (and useful) value,  $a$ . If a value has not

changed much from the previous value, we have reached the vicinity of a fixed-point. Thus, computers can do a very good approximation to a function's fixed-point. **The initial value is crucial, however.** For example, for the function  $x^2$ , had we started with  $a = 0.5$ :

$$f(a) = (0.5)^2 = 0.25$$

$$f(f(a)) = (0.25)^2 = 0.0625$$

$$f(f(f(a))) = (0.0625)^2 = 0.00390625$$

$$f(f(f(f(a)))) = (0.00390625)^2 = 0.00001525878$$

This value approaches 0, which is indeed a fixed-point of  $x^2$ . We could find this fixed-point also if we start with an initial guess,  $a = 0$ .

We could find the other fixed-point ( $a = 1$ ), only if we were lucky to start with an initial guess,  $a = 1$ . But if we started with any other number greater than 1, repeated application of our function will only take us farther and farther away from both the fixed-points of this function,  $f(x) = x^2$ . Also, if we start with the initial guess that is less than 1, we can only get to 0, but never to 1. This is quite problematic, isn't it?

How do we know a priori if an initial guess is *useful* in finding a fixed-point of **any**  $f$  whose fixed-point exists? I am sure there is enough research done on this problem already. A quick glance at the fixed-point page on Wikipedia demonstrates that.

Well, for **periodic functions** (like sine or cosine) that are defined for all real numbers, there's hope. But it is far from clear if the procedure of repeatedly finding the value of a real function (when starting from some initial guess) will converge to its fixed-point. The iterative procedure above is called the **Fixed-point Iteration**. The Wikipedia [page](#) on it says

More specifically, given a function  $f$  defined on the **real numbers** with real values and **given a point**  $x_0$  in the **domain** of  $f$ , the fixed-point iteration is

$$x_{n+1} = f(x_n), n = 0, 1, 2, \dots$$

which gives rise to the **sequence**  $x_0, x_1, x_2, \dots$  of **iterated function** applications

$x_0, f(x_0), f(f(x_0)), \dots$  which is **hoped to converge** to a point  $x_{FIX}$ . If  $f$  is continuous, then one can prove that the obtained  $x_{FIX}$  is a fixed point of  $f$ , i.e.,

$$f(x_{FIX}) = x_{FIX}.$$

As you can see, we have to be lucky in choosing  $x_0$ . If the sequence of repeated application of  $f$  of values starting with  $x_0$  does not converge, then we may never find the function's fixed-point!

First, let's take a stab at the fixed-point "method". This is a method because it generally applies to other functions for which a fixed-point is known to exist (we'll discuss later what to do with functions that do not have a fixed-point). Here is its specification:

- 1) It applies to two formal parameters,  $f$  and  $guess$ .
- 2) If  $f(guess) \sim guess$ , it returns  $guess$ .
- 3) Otherwise it applies itself to  $f$  and  $f(guess)$  and goes back to step 2).

```
const tolerance = 0.0001; // some small value
function fixed_point(f, guess) {
  function close_enough(a, b) {
    return math_abs(a - b) < tolerance;
  }
  function try_with(guess) {
    const next = f(guess);
    return close_enough(guess, next)
      ? next // one may return guess
      : try_with(next);
  }
  return try_with(guess);
}
```



Now, given what we do in the "method" of fixed points (that is given a function and a guess, we repeatedly apply the function to that guess till the values don't change appreciably) **which function** yields the square-root of a number?

I still don't have a satisfactory answer to this "[inverse problem](#)":

- 1) We have a higher order function `fixed_point(f, guess)`.
- 2) We want to find a function  $f$ , such that `fixed_point(f, guess)` yields  $\sqrt{a}$ .

SICP JS provides this:

In fact, we can readily formulate the square-root computation as a fixed-point search.

Computing the square root of some number  $x$  requires finding a  $y$  such that  $y^2 = x$ .

Putting this equation into the equivalent form  $y = \frac{x}{y}$ , we recognize that we are looking

for a fixed point of the function  $y \rightarrow \frac{x}{y}$  and we can therefore try to compute square roots as

```
function sqrt(x) {
  return fixed_point(y -> x/y, 1);
}
```

This does not pacify me, but that is okay.

Consider finding the square-root of 89. How do I go about devising the  $f$  such that the application of `fixed_point` to that  $f$  and a guess, say, 1: `fixed_point(f, 1)` yields  $\sqrt{89}$ ?

Is it **obvious** that that  $f$  must be  $x \rightarrow \frac{89}{x}$  (in lambda expression form) for `fixed_point(f, 1)` to yield  $\sqrt{89}$ ?

It seems believable, but it is not immediately obvious to me. One clear ray of hope is that if the function  $f(x) = \frac{89}{x}$  is applied to  $x = 9.4$ , then it returns 9.47 which is close to  $x$ . If  $f$  is applied to  $x = 9.42$ , it returns 9.45 and so on. At some point, the repeated application is expected to yield a value that is very close to the input. That value is  $\sqrt{89}$ . If applied to any other value of  $x$ , clearly,  $f$  will return a value that is *considerably* different from it. Thus, the transformation  $y = \frac{89}{x}$  of  $y = \sqrt{89}$  seems to do the trick. Of course, if the function does the trick for 89, it will do it for any other positive real number.

In this sense, the function `fixed_point` acts like a *loop* of a special kind: **It keeps on applying  $f$  as long as  $f$ 's input (whose initial value is an appropriate guess) and output are considerably different values; it terminates when they are close enough.**

Thus,  $f(x) = \frac{89}{x}$  is the function to which (and to a guess) if we apply our `fixed_point` *method*, we get an approximate value of  $\sqrt{89}$ .

**However, this function call: `fixed_point(y => x/y, 1)` never returns on many inputs; it's stuck in an infinite loop.** This happens because (when applied to the initial guess 1):  $f(1) = 89$  and  $f(89) = 1$ ! And since 89 and 1 are not close enough, this sequence of return values – 89, 1, 89, 1, ... – continues ad infinitum. The ***fixed-point iteration or fixed-point search*** (that is what the iteration represented by the repeated function calls of `fixed_point` is called) does not terminate.

This is a big problem! What appears sound in theory fails miserably in the practice of numerical computation. One should refer to a good book or article on numerical analysis because now we need to find an algorithm that works on real numbers. The book says that since

$f(x) = y = \frac{89}{x}$  fails to “converge” to some value, we need to find another function which 1) finds the square root of 89, and 2) converges more quickly.

A trick is suggested:  $y = \frac{89}{x} \rightarrow 2y = y + \frac{89}{x} \rightarrow y = \frac{1}{2} (y + \frac{89}{x})$ . This is called the “average-damping.” Plots of both these functions ( $y = \frac{89}{x}$ ,  $y = \frac{1}{2} (y + \frac{89}{x})$ ) expectedly look identical. The latter converges quickly, whereas the former does not converge at all. The `sqrt` function then becomes:

```
function sqrt(x) {  
  return fixed_point(y => average(y, x/y), 1);  
}
```

I am pretty sure that there are reasons for this (why a slightly modified version of the function “converges” quickly) that are studied in numerical analysis.

21 June 2023

### 1.3.2 Functions as Lambda Expressions (LE's)

***We can use a lambda expression as a syntactic form for creating functions.***

In general, lambda expressions are used to create functions in the same way as function declarations, except that ***no name is specified for the function*** and the `return` keyword and braces are omitted (if there is only one parameter, the parentheses around the parameter list can also be omitted). LE's can be extended to allow a block (rather than a mere expression) as the body:

```
(parameters) => expression
```

Whatever results is a function, except that there is no name to associate it with. Therefore, it can either be passed as a value wherever a function value is needed, or it can be applied ***in place*** to actual arguments. In such cases the construct is called a ***function expression***:

```
((x, y, z) => (x + y + z))(1, 2, 3); //=> 1 + 2 + 3 = 6
```

To me, the main advantage of LE's is to aid readability by reducing the scope of function's definition: you don't need to look elsewhere to find out what procedure a named function stands for. Of course, they are best suited in situations where expressions, rather than a full-fledged process, are needed.

Using `const` to create local names

There are some subtleties here, but basically, we can use the `const` keyword to create functions via LE's.

## Conditional Statements

Sometimes conditional statements are better than conditional expressions. We have been using [conditional expressions](#) forthwith. The general form of conditional statements is:

```
if (predicate) {  
    consequent-statements  
} else {  
    alternative-statements  
}
```

### [Exercise 1.34](#)

Suppose we declare

```
function f(g) {  
    return g(2);  
}
```

Then we have

```
f(square);  
4  
f(z => z * (z + 1));  
6
```

What happens if we (**perversely** (in a manner contrary to what is expected or accepted)) ask the interpreter to evaluate the application `f(f)`? Explain.

### **Solution.**

Initially, I thought that this would result in endless application of `f`. But on careful analysis, I realized that `f` is written to accept a function that accepts a single number. A function that accepts a single number – let's call it `g`. Thus, `f` can be applied successfully to any `g`. Is `square` a `g`? Yes, it is. Is `inc` a `g`? Yes.

Is `f` a `g`? No, `f` is `f`. So, applying `f` to `f` is going to result in something undesirable:

`f(f) => f(2) => 2(2)`. Clearly, `2` is not a function. The actual error that the Source-1 interpreter emits is: **Calling non-function value 2.**

20 June 2023

(I had taken a short break from regular work on SICP.)

### [Exercise 1.32](#)

#### **1.32.1**

Show that `sum` and `product` (exercise [1.31](#)) are both special cases of a still more general notion called `accumulate` that combines a collection of terms, using some general accumulation function:



```
accumulate(combiner, null_value, term, a, next, b);
```

The function `accumulate` takes as arguments the same term and range specifications as `sum` and `product`, together with a `combiner` function (of two arguments) that specifies how the current term is to be combined with the accumulation of the preceding terms and a `null_value` that specifies what base value to use when the terms run out. Write `accumulate` and show how `sum` and `product` can both be declared as simple calls to `accumulate`.

### Solution.

As you wander in the enchanted land of functional programming, you can't escape the feeling that you can generalize as much as you want to. I guess the ulterior motif is to teach how to create more abstractions. It seems to me, however, that a balance should be achieved, otherwise you may have functions that are too general to be usable and readability of code may suffer. I wonder if there is an objective way to determine the limits on generalizing abstract ideas or is it a matter of subjective notions like beauty? Perhaps we can exploit the famous maxim to elucidate: Make everything as abstract as possible, but no more. It appears to be a matter of experience that enables seeing how everything fits together.

Let's take a look at the `sum` and `product` functions and find places where we can move from **concrete** to **abstract**.

<code>sum(a, term, next, b)</code>	<code>product(a, term, next, b)</code>
<pre>function <b>sum</b>(a, term, next, b) {   return a &gt; b     ? <b>0</b>     : term(a) + <b>sum</b>(next(a), term, next, b); }</pre>	<pre>function <b>product</b>(a, term, next, b) {   return a &gt; b     ? <b>1</b>     : term(a) * <b>product</b>(next(a), term, next, b); }</pre>

We can see that the elements in **boldface** (`sum`, `0`, `+` and `product`, `1`, `*`) are concrete. This may become a hint in conceiving a higher-level of abstraction that covers both the finite sum and finite product notions (in mathematical notation,  $\sum$ ,  $\prod$ ) respectively. Such a higher-level of abstraction may even cover some more operations that we can name (just like we have named here – one operation “sum” and another “product”). But can't we abstract out (via the notion of parameters) the *summingness* or *productness*? Perhaps we can. This notion or idea can then be captured in the English verb, say, **accumulate**.

Perhaps we can introduce the idea of **combining** a next term (realized by the `term` function) with the result we have up until now to produce a new result. So, a **combiner** function does just

that. We can pass those two *parameters* to `combiner` which returns the result that we return from our function.

What about the two values 0 and 1 which seem to make sense with `sum` and `product` respectively?

Clearly, if we are asked to sum integers *from*, say, 10 *to* 8, we assume that it is 0. No other result makes sense:

“What is the sum of all the integers from  $a$  to  $b$  where  $a > b$ ?”

“Of course it is 0.”

With the *product* operation (on numbers), what value makes sense when multiplying all the integers from  $a$  to  $b$  where  $a > b$ ? This is taken to be 1 by convention. It’s called the [empty product](#): The product of all numbers when there are no numbers to multiply is 1.

What we do here is abstract this difference out. We look at `sum` and `product` from a higher level of abstraction so that they look like *different manifestations of the same thing* to us! A definition then emerges:

```
// accumulation as an abstraction
function accumulate(combiner, null_value, term, a, next, b) {
  return a > b
    ? null_value
    : combiner(term(a), accumulate(combiner, null_value, term,
next(a), next, b));
}
```

To define `sum` in the form of `accumulate` we will need an *adder*:

```
function add(a, b) {
  return a + b;
}
function sum(term, a, next, b) {
  return accumulate(add, 0, term, a, next, b);
}
```

Equivalently, for `product`, we need a function like `multiply`:

```
function multiply(a, b) {
  return a * b;
}
function product(term, a, next, b) {
  return accumulate(multiply, 1, term, a, next, b);
}
```

Watch out for the “telescoping” function, however (a function that applies to several arguments).

### 1.32.2

If your `accumulate` function generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

### Solution.

Our `accumulate` function returns what `combiner` returns, but `combiner` applies to the return value of a recursive call to `accumulate`. Clearly, there are deferred operations that result in a recursive process. We want an iterative version of `accumulate`.

Doing computations on the fly is key. For that, we can employ an accumulator or collector. In this case, we do not need the `null_value` argument because the accumulator serves the purpose.

```
// accumulates results, creates an iterative process
function accumulate_iter(combiner, term, a, next, b, acc) {
  return a > b
    ? acc
    : accumulate_iter(combiner, term, next(a), next, b,
combiner(acc, term(a)));
}
```

(This took some experimentation to get right). Here, we calculate the sum and product directly, without providing the so called wrapper functions that wrap the `accumulate` function:

```
accumulate_iter(add, identity, 1, inc, 10, 0); // => 55
accumulate_iter(multiply, identity, 1, inc, 5, 1); // => 120
```

---

Is `accumulate` or `accumulate_iter` a better abstraction<sup>1</sup> (than `sum` and `product`)?

In a way, it's a matter of perspective and language. When we think of a “summing operation”, do we want to define a specific `sum` function that we *call* (that is, apply to arguments)? If all we want to do is calculate sums, then it is perhaps okay. But what if we want to calculate products? Do we then conceive a separate `product` function which is almost identical to the `sum` function? At a linguistic level, is the code easier to understand if we have those words, “sum” and “product”, in function definitions? If yes, like we have done above (in the `accumulate` case), we provide wrapper functions. If not, we just apply the functions to appropriately chosen arguments (as in the `accumulate_iter` case above; we did not wrap `accumulate_iter` in a `sum_iter` or `product_iter` function, but simply applied it).

It really depends. We should strive hard to make our code easier to read. This is why we choose appropriate names. A function application (function call) like:

```
accumulate_iter(add, identity, 1, inc, 10, 0); // => 55
```

---

<sup>1</sup> `accumulate` and `accumulate_iter` are not different abstractions, just different implementations of the same idea.

makes it clear that this returns 55 because of appropriately named function arguments like `add`, `identity`, `inc`, although it makes it less clear that 0 is the accumulator that stores the result. But what if we had named functions in a weird manner? What value does `accumulate_iter(foo, bar, 1, f, 10, 0); return?`

Do feelings come into picture when faced with deciding if this is a “good” abstraction? Perhaps they do. Our current **feeling** is that `accumulate` is definitely a useful abstraction that is at a higher level than `sum` and `product` which are themselves good abstractions.

### Exercise 1.33

You can obtain an even more general version of `accumulate` (exercise 1.32) by introducing the notion of a **filter** on the terms to be combined. That is, combine only those terms derived from values in the range **that satisfy a specified condition**. The resulting `filtered_accumulate` abstraction takes the same arguments as `accumulate`, together with an **additional predicate of one argument** that specifies the filter. Write `filtered_accumulate` as a function. Show how to express the following using `filtered_accumulate`:

- 1) the sum of the squares of the prime numbers in the interval  $[a, b]$  (assuming that you have an `is_prime` predicate already written).
- 2) the product of all the positive integers less than  $n$  that are relatively prime to  $n$  (i.e., all positive integers  $i < n$  such that  $GCD(i, n) = 1$ ).

### **Solution.**

This is more like a convenience. The main idea is to generate the next item using the `next` function, but have flexibility of filtering it out. One may then ask whether `next` can do it. Perhaps it can. For example, if we want to only sum odd integers in the given range,  $[a, b]$ , we can always generate the next odd integer, rather than generating the next integer.

Here it becomes a matter of taste. Filtering can be achieved indirectly by the implementation of the `next` function, but an explicit predicate (appropriately named `filter`) makes the idea clearer to the readers.

```
function filtered_accumulate(combiner, null_value, term, a, next, b,
filter_in) {
  return a > b
    ? null_value
    : filter_in(a)
      ? combiner(term(a), filtered_accumulate(combiner, null_value,
term, next(a), next, b, filter_in))
      : filtered_accumulate(combiner, null_value, term, next(a),
next, b, filter_in);
}
```

```

function is_even(a) {
    return a % 2 === 0;
}

function is_odd(a) {
    return ! is_even(a);
}

filtered_accumulate(add, 0, identity, 1, inc, 5, is_even); // => 2 +
4 = 6
filtered_accumulate(add, 0, square, 1, inc, 5, is_odd); // => 1x1 +
3x3 + 5x5 = 35

```

- 1) The sum of the squares of the prime numbers in the interval  $[a, b]$  =  
`filtered_accumulate(add, 0, square, a, inc, b, is_prime);`
- 2) the product of all the positive integers less than  $n$  that are relatively prime to  $n$  (i.e., all positive integers  $i < n$  such that  $GCD(i, n) = 1$ ):

```

function gcd(a, b) {
    return b === 0
        ? a
        : gcd(b, a % b);
}

function rel_prime(a, b) {
    return gcd(a, b) === 1;
}

function rel_prime_prod(n) {
    function is_rel_prime(i) {
        return rel_prime(n, i);
    }
    return filtered_accumulate(multiply, 1, identity, 1, inc, n,
is_rel_prime);
}

display(rel_prime_prod(6)); // => 5
display(rel_prime_prod(10)); // => 189

```

08 June 2023

[Exercise 1.31](#)

### 1.31.1

The `sum` function is only the simplest of a vast number of similar abstractions that can be captured as higher-order functions.[3](#) Write an analogous function called `product` that returns the product of the values of a function at points over a given range. Show how to define factorial in terms of `product`. Also use `product` to compute approximations to  $\pi$  using the formula [4](#):

$$\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 6 \cdot 8 \cdots}{3 \cdot 5 \cdot 7 \cdots} \text{ (this is a misleading representation.)}$$

#### Solution.

```
function product(a, term, next, b) {
  return a > b
    ? 1
    : term(a) * product(next(a), term, next, b);
}
```

```
function factorial(n) {
  return product(1, identity, inc, n);
}
```

```
factorial(12); // => 479001600
```

One can observe that the Wallis formula can be rearranged to yield:

$$\frac{\pi}{4} = 2 \cdot \left(\frac{4}{3}\right)^2 \cdot \left(\frac{6}{5}\right)^2 \cdot \left(\frac{8}{7}\right)^2 \cdot \left(\frac{10}{9}\right)^2 \cdots \text{-----} \quad (1.31.1.1)$$

Thus, the denominator of each fraction can be put into a 1:1 correspondence with the natural numbers:  $2n + 1$ . This is an infinite series that converges to  $\pi/4$ , but never equals (and, hence, never exceeds) it. We accept a parameter that specifies the number of fractions to consider in this product:

```
function wallis_pi(n) {
  function term(i) {
    return (2 * i + 2) / (2 * i + 1);
  }
  return 2 * product(1, term, inc, n); // should approach  $\pi/4$ 
}
```

```
wallis_pi(10) ; // => 17.675817171612533
```

Say what? How did we go wrong?

This reminds me of a famous quote: Every problem has a solution that is simple, elegant, and wrong!

Without even realizing (until we ran the program that faithfully and correctly calculated what it was asked to) we have committed a huge blunder.

Infinite series bite us subtly (that is why I need to study it more thoroughly). That's what happened. We rather coolly wrote in equation 1.31.1.1 what John Wallis never intended. Wikipedia [describes](#) what Wallis wrote:

$$\frac{\pi}{2} = \prod_1^{\infty} \frac{2n}{2n-1} \cdot \frac{2n}{2n+1} = \prod_1^{\infty} \frac{4n^2}{4n^2-1}$$

(Notice how the rearrangement of  $\frac{\pi}{4}$  to  $\frac{\pi}{2}$  is allowed, but 1.31.1.1 is not. Exactly what prevents us from doing that is a question that I need to think about. **TODO.**)

Once the confusion is cleared, we can easily code the Wallis product:

```
function wallis_pi_corrected(n) {  
  function term(i) {  
    const numerator = 4 * square(i);  
    return numerator / (numerator - 1);  
  }  
  return 2 * product(1, term, inc, n); // should approach  $\pi$   
}
```

```
wallis_pi_corrected(10);    // => 3.0677038066434994  
wallis_pi_corrected(100);  // => 3.133787490628163  
wallis_pi_corrected(1000); // => 3.1408077460304042
```

This appears more correct. One can just observe this formula with amazement and feel great

about it! The series  $\prod_{n=1}^{\infty} \left(\frac{2n+2}{2n+1}\right)^2$  diverges, but  $\prod_{n=1}^{\infty} \frac{4n^2}{4n^2-1}$  converges to  $\pi/2$ !

### 1.31.2

If your `product` function generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

#### **Solution.**

We use the standard accumulator trick that comes in handy when converting a recursive process into an iterative one.

```
function product_iter(a, term, next, b, acc) {
```

```

    return a > b
        ? acc
        : product_iter(next(a), term, next, b, acc * term(a));
}

```

```

function factorial_iter(n) {
    return product_iter(1, identity, inc, n, 1);
}

```

```
factorial_iter(12); // => 479001600
```

07 June 2023

### [Exercise 1.30](#)

The `sum` function above generates a linear recursion. The function can be rewritten so that the sum is performed iteratively. Show how to do this by filling in the missing expressions in the following declaration:

```

function sum(term, a, next, b) {
    function iter(a, result) {
        return <??>
            ? <??>
            : iter(<??>, <??>);
    }
    return iter(<??>, <??>);
}

```

### **Solution.**

```

function sum_iter(a, term, next, b) {
    function iter(curr, result) {
        return curr > b
            ? result
            : iter(next(curr), result + term(curr));
    }
    return iter(a, 0);
}

```

The idea is to use the argument `result` as an *accumulator*.



05 June 2023

Building abstractions that use functions as arguments is a fun exercise. Thinking of the right abstractions that are reusable in a variety of situations takes practice. In our present case, we have a *summing abstraction* that easily implements integral (Riemann sum).

### [Exercise 1.29](#)

Simpson's Rule is a more accurate method of numerical integration than the method illustrated above. Using Simpson's Rule, the integral of a function  $f$  between  $a$ ,  $b$  is approximated as

$\frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \dots + 2y_{n-2} + 4y_{n-1} + y_n)$ , where  $h = \frac{a-b}{n}$  for some even

integer  $n$ , and  $y_k = f(a + kh)$ . (Increasing  $n$  increases the accuracy of the approximation.)

Declare a function that takes  $f$ ,  $a$ ,  $b$ ,  $n$  as arguments and returns the value of the integral, computed using Simpson's Rule. Use your function to integrate `cube` between 0 and 1 (with  $n = 100, 1000$ ), and compare the results to those of the `integral` function shown above.

### Solution.

Simpson's rule is an interesting approximation to find definite integrals of continuous functions. It approximates the area under the curve  $f(x)$  from  $x = a$  to  $x = b$ . Fitting the computation in the abstraction provided by the `sum` function is not that hard. Let's first do that and then we'll see if other, generally usable abstractions emerge.

```
function simpson(f, a, b, n) {
  const h = (b - a) / n;
  function s_next(t) {
    return t + h;
  }
  return sum(a, f, s_next, b) * h / 3;
}
```

**This definition will simply calculate each term as  $f(a)$  which is *not* what we want.** We have the sum as:  $f(a) + 4f(a + h) + 2f(a + 2h) + \dots + 4f(a + (n - 2)h) + f(a + (n - 1)h)$ . Thus, depending on the rank (0, 1, 2, 3, ...,  $n$ ) of the term, its value changes. Wikipedia reports this succinctly:

$$\begin{aligned} \int_a^b f(x) dx &\approx \frac{1}{3}h \sum_{i=1}^{n/2} [f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i})] \\ &= \frac{1}{3}h [f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \dots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)] \\ &= \frac{1}{3}h \left[ f(x_0) + 4 \sum_{i=1}^{n/2} f(x_{2i-1}) + 2 \sum_{i=1}^{n/2-1} f(x_{2i}) + f(x_n) \right]. \end{aligned}$$

We need to somehow introduce the notion of saving the context (which term is it that we are calculating in a given invocation of `term`?). Without much clarity, I coded the following. I went wrong once. **But after a good night's sleep (as the sleep was getting over), I told myself that I should have a notion of an *index* built into the summing operation:**

```
function sum_with_index(a, term, next, b, i) {
  return a < b
    ? 0
    : term(a, i) + sum_with_index(next(a, i), term, next, b, i +
1);
}
```

This is perhaps not enough because it may be a good idea to know the maximum value the index (it always starts at 0) can take.

```
// sums up given terms and maintains an index
function sum_with_index(a, term, next, b, i, maxi) {
  return a < b
    ? 0
    : term(a, i, maxi) + sum_with_index(next(a, i, maxi), term,
next, b, i + 1, maxi);
}
```

We can now attempt the Simpson  $\frac{1}{3}$  rule and the Riemann sum:

```
// sums the terms returned by term(a) as long as a <= b
function sum(a, term, next, b) {
  return a > b
    ? 0
    : term(a) + sum(next(a), term, next, b);
}

// returns the riemann sum for a given function (continuous)
function riemann_sum(f, a, b, dx) {
  function add_dx(x) {
    return x + dx;
  }
  function term(x) {
    return f(x);
  }
  return dx * sum(a + dx / 2, term, add_dx, b);
}

// implements the Simpson 1/3 rule
function simpson(f, a, b, n) {
  const h = (b - a) / n;
  function term(a, i, maxi) {
    return i === 0 || i === maxi
```

```

        ? f(a)
        : i % 2 === 1
        ? 4 * f(a) // odd
        : 2 * f(a); // even
    }
    function next(t, i, maxi) {
        return t + h; // i, maxi unused
    }
    return h / 3 * sum_with_index(a, term, next, b, 0, n);
}

function cube(x) {
    return x * x * x;
}

function inc(x) {
    return x + 1;
}
// sums the cubes of integers between a and b
function sum_cubes(a, b) {
    function cube_with_index(x, i, maxi) {
        const c = x * x * x;
        display(x, c, i, maxi);
        return c;
    }
    function next(x, i, maxi) {
        return x + 1;
    }
    return sum_with_index(1, cube_with_index, next, 5, 0, 5);
}
// sum_cubes(1, 5);
display(riemann_sum(cube, 0, 1, 0.001)); //=> 0.249999875000001
display(simpson(cube, 0, 1, 100)); // => 0.246666666666666687
display(simpson(cube, 0, 1, 1000)); // => 0.249666666666666754

```

It's not immediately clear that Simpson's  $\frac{1}{3}$  rule always gives a more accurate answer. I also suspect that there is something wrong with the implementation. **TODO**.

**Is `sum_with_index` a good abstraction? Does it add value above and beyond `sum`?** I discussed this with Apoorv and he was able to come up with a much better idea of reusing the same abstraction and deriving the context in the implementation of the `term` function! I am now somewhat distraught that this did not occur to me. So, the `sum` function arranges for a counter to loop through and then the `term` function uses that counter as the index:

```
function simpson(f, a, b, n) {
  // some supporting code
  return sum(0, term, inc, n); // this is clever, loops 0->n
}
```

We leave  $a$ ,  $f$  in scope, but provide the counters 0 and  $n$  in the sum function invocation. This solves the problem of providing the context.

```
function simpson_improved(f, a, b, n) {
  //f, a, b, n are all in scope!
  const h = (b - a) / n;
  function y(k) {
    return f(a + k * h);
  }
  function term(i) {
    return i === 0 || i === n
      ? y(i)
      : i % 2 === 1
      ? 4 * y(i)
      : 2 * y(i);
  }
  return (h / 3) * sum(0, term, inc, n);
}
simpson_improved(cube, 0, 1, 1000); // => 0.2500000000000003
```

This yields a much more accurate answer. Why would there be

01 June 2023

### 1.3 Higher Order Functions

One of the things we should demand from our programming language is the ability to build abstractions by assigning names to common patterns and then to work directly on those abstractions.

Functions provide this ability.

**Functions that manipulate functions are called higher-order functions.** Higher-order functions serve as powerful abstraction mechanisms, vastly increasing the expressive power of our programming language.

### 1.3.1 Functions as Arguments

Consider these three functions that do summing computations for each integer in a given range  $a, b$  (inclusive):

```

$$// \sum_{i=a}^b i$$
  
function sum_integers(a, b) {  
  return a > b  
    ? 0  
    : a + sum_integers(a + 1, b);  
}  
  

$$// \sum_{i=a}^b i^3$$
  
function sum_cubes(a, b) {  
  return a > b  
    ? 0  
    : a * a * a + sum_cubes(a + 1, b);  
}  
  

$$// \sum_{i=a}^b \frac{1}{(4i-3) \cdot (4i-3)}$$
 which slowly converges to  $\frac{\pi}{8}$ .  
function sum_pi8(a, b) {  
  return a > b  
    ? 0  
    : 1 / ((4 * a - 3) * (4 * a - 1)) + sum_pi8(a + 1, b);  
}
```

Color-coding reveals interesting things about this “procedure”. One can observe a pattern with the code of these functions. The only thing that differs in these recursive functions is the values that we are actually summing up (shown in blue)! These values can easily be thought of as functions of the current value of the variable of summation.

The last function can be rewritten as:

```

$$// \sum_{i=a}^b \frac{1}{(4i-3) \cdot (4i-3)}$$
 which slowly converges to  $\frac{\pi}{8}$ : alternate method.  
function sum_pi8(a, b) {  
  return a > b  
    ? 0  
    : 1 / (a * (a + 2)) + sum_pi8(a + 4, b);  
}
```

This is the idea behind summation using the well-known *Sigma notation*:

$$\sum_{i=a}^b f(i) = f(a) + f(a + 1) + \dots + f(b).$$

We can identify two items of interest:

- 1) The addend corresponding to the current value of the variable of summation, and
- 2) The next value of the variable of summation. In classical sigma notation, this follows the natural numbers in that the next value is just the next integer. But conceptually it need not be so.

We can then make these behaviors into functional parameters. These parameters are like numbers, except that since they are functions, they can, unlike numbers, be ***applied to*** other values.

```
function sum(a, term, next, b) {  
  return a > b  
    ? 0  
    : term(a) + sum(next(a), term, next, b);  
}
```

It just works out!

We can now define some specific functions that work with this higher-order function, `sum` and together we redefine the `sum_integers` function and other functions.

```
function inc(a) {  
  return a + 1;  
}  
  
function identity(a) {  
  return a;  
}  
  
function cube(a) {  
  return a * a * a;  
}
```

Then, `sum_integers` becomes:

```
function sum_integers(a, b) {  
  return sum(a, identity, inc, b);  
}  
  
function sum_cubes(a, b) {
```

```

    return sum(a, cube, inc, b);
}

function sum_pi8(a, b) {
    function pi_term(a) {
        return (1 / (a * (a + 2)));
    }
    function pi_next(a) {
        return a + 4;
    }
    return sum(a, pi_term, pi_next, b);
}

```

The `sum` function can be utilized to express integration which is the riemann sum for continuous functions:

$$\int_a^b f = [f(a) + f(a + \frac{dx}{2}) + f(a + dx + \frac{dx}{2}) + f(a + 2dx + \frac{dx}{2}) + \dots + f(a + (n - 1)dx + \frac{dx}{2})] \cdot dx$$

where  $dx$  is *small*.

```

function integral(f, a, b, dx) {
    function add_dx(x) {
        return x + dx / 2;
    }
    return sum(a + dx / 2, f, add_dx, b) * dx;
}

```

31 May 2023

### [Exercise 1.28](#)

One variant of the Fermat test that cannot be fooled is called the *Miller–Rabin test* (Miller 1976; Rabin 1980). This starts from an alternate form of Fermat's Little Theorem, which states that if  $n$  is a prime number and  $a$  is any positive number less than  $n$ , then  $a^{n-1} \equiv 1 \pmod{n}$ .

To test the primality of a number  $n$  by the *Miller–Rabin test*, we pick a random integer  $a < n$  and we find that  $a^{n-1} \equiv 1 \pmod{n}$ . (We prove this result below).

We use the `expmod` function to calculate the result of the exponentiation  $a^{n-1}$ .

However, **whenever we perform the squaring step** in `expmod`, we check to see if we have discovered a “nontrivial square root of 1 (modulo  $n$ )”, that is a number not equal to 1 or  $n - 1$  and whose square is equal to 1 *modulo*  $n$ .

**It is possible to prove that if such a nontrivial square root of 1 modulo  $n$  exists then  $n$  is not prime.**

It is also possible to prove that if  $n$  is an odd number that is not prime, then, for at least half the numbers  $a < n$ , computing  $a^{n-1}$  in this way will reveal a nontrivial square root of 1 *modulo*  $n$ . (This is why the Miller-Rabin test cannot be fooled.)

Modify the `expmod` function to signal if it discovers a nontrivial square root of 1 and use it to implement the Miller-Rabin test with a function (analogous to the function `fermat_test`). Check your function with various known primes and nonprimes.

### Solution.

In mathematics, as in life, we can discover truths only by building on top of other truths. Thus, if Fermat's Little Theorem (FLT) is assumed, we should be able to prove the basis of the Miller-Rabin test (MRT). Thus, given

- 1) a prime number  $n$ ,
- 2) FLT (that is,  $a^n \equiv a \pmod{n}$ ), and
- 3) an integer  $a < n$

can we prove that  $a^n \equiv 1 \pmod{n}$ ?

### Proof.

For an integer  $a < n$ , the division algorithm tells us that  $a = 0 \cdot n + a$  which means  $a \pmod{n} = a$ .

From division algorithm and FLT it follows that  $a^n = n \cdot q + a$  ----- (1.28.1).

Since  $a|a^n$  and  $a|a$ , it follows that  $a|(n \cdot q)$ , but since  $a < n$ ,  $a \nmid n$ ,  $a|q$ .

Dividing both sides of 1.28.1 by  $a$  we get:

$$a^{n-1} = \frac{q}{a} \cdot n + 1, \text{ that is, } a^{n-1} = q_1 \cdot n + 1, q_1 \in N$$

It then follows that  $a^{n-1} \equiv 1 \pmod{n}$ .

■

The next assertion is more complex. It defines something called a “nontrivial square root of 1 (modulo  $n$ )” (NTSR1): It is an integer  $p < n$ ,  $p \neq 1$ ,  $p \neq n - 1$ , such that  $p^2 \equiv 1 \pmod{n}$ .



Clearly,  $1^2 = 1 \pmod{n}$ . Also, since  $(n - 1)^2 = n^2 - 2n + 1$ ,  $(n - 1)^2 \equiv 1 \pmod{n}$ . But excluding these two integers (1,  $n - 1$ ), do we have any other integers less than  $n$  that are congruent to 1 (modulo  $n$ )? And if we do, then  $n$  is non prime.

I am going to leave that proof as a **TODO** and instead assume the result for now.

Thus, we need to check in our implementation if we have found a number less than  $n$  that is an NTSR1. If we found such an integer, then  $n$  is not prime.

```
// a is an ntsr1 if a != 1, a != (n-1) and a^2 % n = 1
function is_ntsr1(a, n) {
    return a != 1 && a != (n - 1) && square(a % n) === 1;
}
```

So, the procedure seems clear now. Let's call our new primality test function `mr_test` which accepts an integer  $n$ .

We reproduce the [expmod](#) function here:

```
// returns (base)^exp % divisor
function expmod(base, exp, divisor) {
    return exp === 0
        ? 1
        : is_even(exp)
        ? square(expmod(base, exp / 2, divisor)) % divisor
        : ((b % divisor) * expmod(base, exp - 1, divisor)) % divisor;
}
```

This exercise asks us to *signal* (in the squaring step) if we have run into a nontrivial square root of 1 (modulo  $n$ ).

Shouldn't this work?

```
function expmod_w_ntsr1(base, exp, divisor) {
    return exp === 0
        ? 1
        : ! is_even(exp)
        ? ((b % divisor) * expmod(base, exp - 1, divisor)) % divisor
        : is_ntsr1(expmod(base, exp / 2, divisor))
        ? 0 // returns 0 to signal that we have an ntsr1
        : square(expmod(base, exp / 2, divisor)) % divisor;
}
```

And then

```
// The Miller-Rabin test: returns true if n is prime, false otherwise
```

```
function mr_test(n) {
  function try_it(a) {
    return expmod_w_ntsrl(a, n - 1, n) === 1;
  }
  return try_it(1 + math_floor(math_random() * (n - 1)));
}
```

## TODO

19 May 2023

### Exercise 1.26

Louis Reasoner is having great difficulty doing exercise [1.24](#). His `fast_is_prime` test seems to run more slowly than his `is_prime` test. Louis calls his friend Eva LuAtor over to help. When they examine Louis's code, they find that he has rewritten the `expmod` function to use an explicit multiplication, rather than calling `square`:

```
function expmod(base, exp, m) {
  return exp === 0
    ? 1
    : is_even(exp)
      ? (expmod(base, exp / 2, m)
         * expmod(base, exp / 2, m)) % m
      : (base * expmod(base, exp - 1, m)) % m;
}
```

"I don't see what difference that could make," says Louis.

"I do", says Eva. "By writing the function like that, you have transformed the  $\Theta(\log n)$  process into a  $\Theta(n)$  process."

Explain.

### Solution.

Clearly, Louis is calling the function twice (albeit with a halved argument) and multiplying the results. Thus, one problem that accepts an input of size  $n$  gets transformed into two problems of size  $n/2$  each (and a multiplication operation) and the process continues till we arrive at a base case that simply returns 1. We arrive at a base case that returns immediately, but we arrive at the base case a whopping  $n$  times. The total number of steps then becomes  $\Theta(n)$ .

Using the `square` function instead has an effect similar to that of saving a precomputed result for reuse. This transforms one problem of size  $n$  into a problem of size roughly  $n/2$ : a hallmark of a function that grows logarithmically.

### Exercise 1.27

Demonstrate that the Carmichael numbers listed in footnote (4) really do fool the Fermat test. That is, write a function that takes an integer  $n$  and tests whether every  $a$ ;  $1 < a < n$  is congruent to  $a^n$  and try your function on the given Carmichael numbers.

**Solution.**

```
// a non prime number n is a carmichael number if every integer
// a; 1 < a < n is a fermat liar (that is, it defies FLT test)
function is_carmichael(n) {
    function is_probable_fermat_liar_for_all(i) {
        return i >= n
            ? true
            : expmod(i, n, n) === i
            ? is_probable_fermat_liar_for_all(i + 1)
            : display_return(i, "", false);
    }
    return is_probable_fermat_liar_for_all(2);
}
is_carmichael(561) &&
is_carmichael(1105) &&
is_carmichael(1729) &&
is_carmichael(2465) &&
is_carmichael(2821) &&
is_carmichael(6601); // => true
```

18 May 2023

### Exercise 1.25

Alyssa P. Hacker complains that we went to a lot of extra work in writing `expmod`. After all, she says, since we already know how to compute exponentials, we could have simply written

```
function expmod(base, exp, m) {
    return fast_expt(base, exp) % m;
}
```

Is she correct? Would this function serve as well for our fast prime tester? Explain.

**Solution.**

She is right in theory, but not in practice. We use a rather simple result in elementary number theory to make sure we can determine  $(base)^{exp} \% divisor$  without really having to calculate  $(base)^{exp}$ . I have written about it [below](#). On a finite precision machine like a computer, overflow is likely to occur much early when raising a number to a power. **It is not immediately clear to me, however, that the remainder will be preserved in spite of the overflow.**

17 May 2023

### [Exercise 1.23](#)

The `smallest_divisor` function shown at the start of this section does lots of needless testing: After it checks to see if the number is divisible by 2 there is no point in checking to see if it is divisible by any larger even numbers. This suggests that the values used for `test_divisor` should not be 2, 3, 4, 5, 6, ... but rather 2, 3, 5, 7, 9, ....

To implement this change, declare a function `next` that returns 3 if its input is equal to 2 and otherwise returns its input plus 2. Modify the `smallest_divisor` function to use `next(test_divisor)` instead of `test_divisor + 1`. With `timed_prime_test` incorporating this modified version of `smallest_divisor`, run the test for each of the 12 primes found in exercise [1.22](#). Since this modification halves the number of test steps, you should expect it to run about twice as fast. Is this expectation confirmed? If not, what is the observed ratio of the speeds of the two algorithms, and how do you explain the fact that it is different from 2?

### **Solution.**

This is a boring exercise 🙄.

I changed my `smallest_divisor` function thus:

```
function smallest_divisor(n) {
  function next(d, slow) {
    return slow
      ? d + 1
      : d === 2
      ? 3
      : d + 2;
  }
  function find_divisor(n, d) {
    return square(d) > n
      ? n
      : divides(n, d)
      ? d
      : find_divisor(n, next(d, true));
  }
  return find_divisor(n, 2);
}
```

This helps me go back and forth. I do find the resulting implementation (`slow = false` => do not examine divisibility for even numbers greater than 2) faster by a factor of almost 2; especially for larger numbers. Here is an output of two sample runs:

```

divisor += 1
100000007
25
100000037
27
100000039
24
divisor += 1 at 2 and divisor += 2 afterword
100000007
12
100000037
11
100000039
12

```

### [Exercise 1.24](#)

Modify the `timed_prime_test` function of exercise [1.22](#) to use `fast_is_prime` (the Fermat method), and test each of the 12 primes you found in that exercise. Since the Fermat test has  $\Theta(\log n)$  growth, how would you expect the time to test primes near 1,000,000 to compare with the time needed to test primes near 1000? Do your data bear this out? Can you explain any discrepancy you find?

**Solution.**

**TODO.**

13 May 2023

### [Exercise 1.21](#)

Use the `smallest_divisor` function to find the smallest divisor of each of the following numbers: 199, 1999, 19999.

**Solution.**

```

199
1999
7

```

### [Exercise 1.22](#)

Assume a primitive function `get_time` of no arguments that returns the number of milliseconds that have passed since 00:00:00 UTC on Thursday, 1 January, [1970](#). The following `timed_prime_test` function, when called with an integer `n`, prints `n` and checks to see if `n` is

prime. If  $n$  is prime, the function prints [three asterisks](#) followed by the amount of time used in performing the test.

```
function timed_prime_test(n) {
  display(n);
  return start_prime_test(n, get_time());
}

function start_prime_test(n, start_time) {
  return is_prime(n)
    ? report_prime(get_time() - start_time)
    : true;
}

function report_prime(elapsed_time) {
  display(" *** "); // return value ignored
  display(elapsed_time);
}
```

Using this `[timed_prime_test]` function, write a function `search_for_primes` that **checks the primality of consecutive odd integers in a specified range**. Use your function to find the three smallest primes larger than 1000; larger than 10,000; larger than 100,000; larger than 1,000,000. Note the time needed to test each prime. Since the testing algorithm has order of growth of  $\Theta(\sqrt{n})$ , you should expect that testing for primes around 10,000 should take about  $\sqrt{10}$  times as long as testing for primes around 1000. Do your timing data bear this out? How well do the data for 100,000 and 1,000,000 support the  $\sqrt{n}$  prediction? Is your result compatible with the notion that programs on your machine run in time proportional to the number of steps required for the computation?

**Solution.** I don't like the way this exercise is constructed. Instead of instructing readers about *how* to do something, they should have just asked *what* to do.

The above function `[timed_prime_test]` is not a well-written one. It returns `undefined` when it detects a prime number! The value returned by the `display` function is ignored. Perhaps this is one problem with functional programming – in FP, we must write functions that represent useful abstractions. We should also strive to write functions that do one thing and do it well. It hasn't happened that way in the above functions. It is also because writing such a function is not that easy.

I am going to attempt to rewrite parts of the above code.

In general, “while I am doing that, can I also do this? – while I am determining if  $n$  is prime, can I also determine how long the process took?” is the question we need to answer effectively.

We want to determine how long a primality test took. Thus, there are two main objectives and a reporting objective:

- Surely determine if the given number is prime.
- Surely calculate the time the process took.
- Optionally report the time; report it only if the number is prime.

Here is our first attempt:

```
// displays d, returns ret
function display_return(something, time, ret) {
  display(something); // ignore display function's return value
  display(time);
  return ret;
}
```

To write a general-purpose timer like this, at the least we need a capability to pass a variable number of arguments to a function because we do not want to be writing versions for each type of function (based on the number of arguments it takes). For example, here's a version for the monadic (functions taking one argument):

```
function report_time_and_return_monadic_bool(f, arg1) {
  const start_time = get_time();
  return f(arg1)
    ? display_return(arg1, get_time() - start_time, true)
    : false;
}
```

We haven't learned this magic yet (it may be covered in later sections of the text). We have also not yet learned how to pass and return functions to other functions. But that seems like the right thing to do.

The time taken by an `isPrime(n)` function call can then be displayed and returned by:

```
report_time_and_return_monadic_bool(is_prime, n);
```

Now we are in a position to find the `count` smallest prime numbers greater than a lower bound and report the time each operation took only when a given number is prime:

```
// searches for as many as count smallest prime numbers greater
// than low (exclusive) , reporting the time spent in each case.
function search_for_primes(low, count) {
  return count <= 0
    ? undefined
    : report_time_and_return_monadic_bool(is_prime, low + 1)
    ? search_for_primes(low + 1, count - 1)
    : search_for_primes(low + 1, count);
}
```

```
}
```

On a typical call like

```
search_for_primes(10000000, 3);
```

it produces an output like:

```
10000019
```

```
6
```

```
10000079
```

```
4
```

```
10000103
```

```
5
```

```
undefined
```

(The ugly `undefined` line is what `search_for_primes` returns at the end of its iteration.)

I think the main problem is mixing the return value types. I guess we'll learn about this later.

It does appear that when the lower bound increases tenfold, the time required increases by a factor that is roughly  $\sqrt{10}$ . But the actual time determination is quite difficult as what happens in silicon at runtime is quite unpredictable. This is the reason one needs to analyze programs and algorithms on paper.

## 11 May 2023

### 1.2.6 Testing for Primality

This is a rich research area: determining if a given number is prime. We know the sieve of Eratosthenes. It is conceivable that we keep running a few computers which generate a giant sieve of Eratosthenes and we can then easily look up in it if a number is prime. But what if we want to know the *prime status* of a number that has not yet figured in the current state of the computation? Clearly, we need more direct methods to determine if a given number is prime.

One way to determine if an integer  $n$  is prime is to find its divisors: If an integer between 1 and  $n$  divides  $n$ , then it is not prime, otherwise it is prime.

If we don't have access to a sieve of Eratosthenes, then there is a rather straightforward way to find the smallest divisor of a given number:

```
function smallest_divisor(n) {  
    return find_divisor(n, 2);  
}
```



```

function find_divisor(n, test_divisor) {
    return square(test_divisor) > n
        ? n
        : divides(n, test_divisor)
        ? test_divisor
        : find_divisor(n, test_divisor + 1);
}

function divides(a, b) { // does b divide a?
    return a % b === 0;
}

function is_prime(n) {
    return n === smallest_divisor(n);
}

```

This algorithm takes  $\Theta(\sqrt{n})$  steps. It is also an iterative process, which means that a tail-call-optimizing interpreter needs constant space to execute it.

## The Fermat Test

The above  $\Theta(\sqrt{n})$ -step algorithm works well, but it takes a lot of time for large numbers. It also wastes time testing primality by dividing the given number by composite numbers. Can we do better?

It turns out that we can give it a try!

A  $\Theta(\log n)$ -step primality test is based on a result from number theory known as Fermat's Little Theorem (FLT):

If  $n$  is a prime number and  $a$  is any positive integer less than  $n$ , then  $a^n \equiv a \pmod{n}$

$a \equiv b \pmod{n}$  is read “ $a$  is **congruent to**  $b$  modulo  $n$ ”. This means that the numbers  $a$  and  $b$  leave the same remainder when divided by  $n$ . In computer terms,  $a \% n = b \% n$ .

FLT is assumed in what ensues, but proving it is a good number theory exercise.

### Proof of the FLT.

**TODO.**

How does knowing “ $a^n \equiv a \pmod{n}$ ” is true for a prime number  $n$  and an integer  $a$ ;  $1 < a < n$ ” help to test if  $n$  is prime?

Suppose we are asked to determine if  $n$  is prime.

We

- 1) raise some random number  $a$  less than  $n$  to the  $n^{\text{th}}$  power ( $a^n$ )
- 2) verify that  $a^n \% n$  indeed equals  $a$ .

Does that make  $n$  prime?

**Example.** Let  $n = 6$ ,  $a = 3$ . Then,  $a^n \% n = 3^6 \% 6 = 729 \% 6 = 3 = a \Rightarrow 3^6 \equiv 3 \pmod{6}$ .

But that does not make 6 a prime number! This is because we are assuming that the *converse* of the FLT is true. Since we found a counterexample, the converse is not true. The theorem only says that if  $n$  is prime, then  $a^n \equiv a \pmod{n}$ . Thus, we need to know that a number  $n$  is prime for the FLT to hold for it.

But we know that a conditional statement is equivalent to its contrapositive. The contrapositive of the FLT's conditional statement is: **If  $a^n \not\equiv a \pmod{n}$ , then  $n$  is not prime.**

Thus, our Fermat's test tells us that for a given  $n$  and a random integer  $a$ ;  $1 < a < n$ ,  $a^n \not\equiv a \pmod{n}$ , then we know for sure that  $n$  is not prime. Otherwise (i.e. if  $a^n \equiv a \pmod{n}$ ), then  $n$  **may be** prime.

We have a test that has no false negatives, but it may have false positives (it may wrongly identify a number as a prime number, but it will never wrongly identify a composite number).

So, how does that test help determine if a given number  $n$  is prime? And that too in  $\Theta(\log n)$  steps?

The answer to the first question refers to a class of algorithms we call "Probabilistic Algorithms".

Consider a large  $n$ . Consider a random integer  $a$ ;  $1 < a < n$ . How **likely** is  $a^n \equiv a \pmod{n}$  when  $n$  is composite? For that composite  $n$ , how likely is it that for another  $a$ ;  $1 < a < n$ ,  $a^n \equiv a \pmod{n}$ ? We'll need to go into number theory, but it turns out that there aren't many such  $a$ 's for which  $a^n \equiv a \pmod{n}$  for a composite  $n$ . Thus for a **composite**  $n$  with a few, fixed number (say 5) of tries for different  $a$ 's, the likelihood of  $a^n \equiv a \pmod{n}$  is not very high. With every "passing test" with a different  $a$ , our **confidence** that  $n$  is indeed a prime number increases. Of course, we can't be sure that it is prime just because all the tests we did passed. This is why one should treat a number adjudged prime by such a test a "probable prime".

Answer to the second question follows from the efficiency of the [fast\\_expt](#) function. But there is a subtlety that we need to be aware of. What we are looking for is a function that raises an integer (`base`) to the power of another integer (`exp`) and returns the remainder after dividing the result of exponentiation by another number (`divisor`). We also want to limit the number of steps this function takes to  $\Theta(\log n)$ . One way to do that is to actually raise  $(base)^{exp}$  and then do the remainder operation. Is there an alternative?

Consider integers,  $a_1, a_2, n > 0$ . We can apply the [division algorithm](#):

$$a_1 = nq_1 + r_1 \text{ ----- (1.2.6.1)}$$

$$a_2 = nq_2 + r_2 \text{ ----- (1.2.6.2)}$$

$$\therefore a_1 a_2 = n(nq_1 q_2 + q_1 r_2 + q_2 r_1) + r_1 r_2 \text{ ----- (1.2.6.3)}$$

Since  $n | (nq_1 q_2 + q_1 r_2 + q_2 r_1)$ ,  $(n(nq_1 q_2 + q_1 r_2 + q_2 r_1)) \% n = 0$  and we get

$$(a_1 a_2) \% n = (r_1 r_2) \% n \text{ ----- (1.2.6.4)}$$

Let  $a_1 = a_2 = b^e$  and  $r_1 = r_2 = r$  (the remainders equal).

Then, (1.2.6.4) simplifies to  $b^{2e} \% n = r^2 \% n = (b^e \% n)^2 \% n$ . (1.2.6.4) is the reason we can verify if  $a^n \equiv a \pmod{n}$  without having to first raise  $a$  to the  $n^{th}$  power.

We can now convert this idea into code. Let's call this function `expmod`:

```
// returns (base)exp % divisor
function expmod(base, exp, divisor) {
    return exp === 0
        ? 1
        : is_even(exp)
        ? square(expmod(base, exp / 2, divisor)) % divisor
        : ((b % divisor) * expmod(base, exp - 1, divisor)) % divisor;
}
```

Since the number of steps reduce by half when `exp` is even, we need the number of steps logarithmic in  $n$ .

Now that we have this function, we can test a given number for primeness:

```
function fermat_test(n) {
    function try_it(a) {
        return expmod(a, n, n) === a;
    }
    return try_it(1 + math_floor(math_random() * (n - 1)));
}
```

If we want to try it a few times and return `true` only if all our attempts pass the test, we can do the following:

```
function is_probable_prime(n, times) {
    return times === 0
```

```

    ? true
    : fermat_test(n)
    ? is_probable_prime(n, times - 1)
    : false;
}

```

It seems that there are some (fortunately pretty rare) numbers, called the Carmichael numbers, that defy the determination of primeness by Fermat's test. These numbers  $n$  are not prime, but Fermat's test passes on them for every  $a < n$ . However, there are some variations on Fermat's test that can not be fooled. See exercises on this section for details.

In contrast to the Fermat test, one can prove that, for any  $n$ , the condition does not hold for most of the integers  $a < n$  unless  $n$  is prime. Thus, if  $n$  passes the test for some random choice of  $a$ , the chances are better than even that  $n$  is prime. If  $n$  passes the test for two random choices of  $a$ , the chances are better than 3 out of 4 that  $n$  is prime. By running the test with more and more randomly chosen values of  $a$  we can make the probability of error as small as we like.

05 May 2023

### 1.2.5 Euclid's GCD Algorithm

This is, according to Knuth, the oldest known non trivial algorithm to find the Greatest Common Divisor of two integers. It follows from the division algorithm which is stated as:

For any two integers,  $a, b, b \neq 0$ , there exist two other integers  $q, r$  such that  $a = bq + r$ , where  $0 \leq r < b$ .

Now, let's consider a number  $g$  that divides both  $a, b$ , then  $a = k_1g$  and  $b = k_2g$  where  $k_1, k_2 \in \mathbb{N}$ . Then, it follows from the division algorithm that  $k_1g = k_2gq + r$  which means that  $r = (k_1 - k_2q)g$  which means  $g|r$  (that is  $g$  divides  $r$ ). Thus, if  $g$  is a common divisor of  $a, b$ , it is a divisor also of  $r$ . Of course, there could be many such common divisors. The greatest of all such common divisors of  $a, b, r$  is called the  $GCD(a, b)$ . Thus,  $GCD(a, b) = GCD(b, r)$ .

If we arrange such that  $a \geq b$ , then  $GCD(a, b) = GCD(b, r)$  will result in calculating GCD of two numbers  $b, r$  that are smaller than  $a, b$ . This process can repeat and since  $r$  is at least one less than  $b$  every time, the remainder gets smaller. The iterative process can only end when  $r = 0$  because for the next iteration, when  $b = 0$ , the division algorithm can not apply. Since every number  $a > 0$  divides both  $a$  and 0, it follows that  $GCD(a, 0) = a$ .

It is easy to express Euclid's Algorithm as a function:

```
function gcd(a, b) {
```

```

    return b === 0 ? a : gcd(b, a % b);
}

```

**Lamé's Theorem:** If Euclid's Algorithm requires  $k$  steps to compute the GCD of some pair, then the smaller number in the pair must be greater than or equal to the  $k^{th}$  Fibonacci number.

**Proof.**

**TODO.**

### Exercise 1.20

The process that a function generates is of course dependent on the rules used by the interpreter. As an example, consider the iterative `gcd` function given above. Suppose we were to interpret this function using **normal-order evaluation**, as discussed in section [1.1.5](#). (The normal-order-evaluation rule for conditional expressions is described in exercise [1.5](#).)

Using the substitution method (for normal order), illustrate the process generated in evaluating `gcd(206, 40)` and indicate the `remainder` operations that are actually performed. How many `remainder` operations are actually performed in the normal-order evaluation of `gcd(206, 40)`? In the applicative-order evaluation?

**Solution.**

In normal-order evaluation, the interpreter, while using the substitution model, does *not* evaluate the function arguments before applying it to them; it simply replaces them verbatim until it is time to evaluate them. Here is the `gcd` function:

```

function gcd(a, b) {
    return b === 0
        ? a
        : gcd(b, a % b);
}

```

Here is how a normal-order interpreter might apply `gcd` to 206 and 40. # denotes the number of times the `remainder` operation is called in the current function application (the process is iterative):

```

gcd(206, 40)
40 === 0 is false (#: 0)
- gcd(40, 206 % 40)
  6 === 0 is false (#: 1)
  - gcd(206 % 40, 40 % (206 % 40))
    4 === 0 is false (#: 2)
    - gcd(40 % (206 % 40), (206 % 40) % (40 % (206 % 40)))
      2 === 0 is false (#: 4)
      - gcd((206 % 40) % (40 % (206 % 40)), (40 % (206 % 40)) %
        ((206 % 40) % (40 % (206 % 40))))

```

```

0 === 0 is true
evaluate (206 % 40) % (40 % (206 % 40)) (#: 7 + 4 = 11)
// => 2

```

The remainder operation is called a total of  $11 + 4 + 2 + 1 = 18$  times.

An applicative-order interpreter evaluates the arguments and then applies the functions to them:

```

gcd(206, 40)
40 === 0 is false (#: 0)
- gcd(40, (206 % 40) = 6) (#: 1)
  6 === 0 is false
  - gcd(6, (40 % 6) = 4) (#: 1)
    4 === 0 is false
    - gcd(4, (6 % 4) = 2) (#: 1)
      2 === 0 is false
      - gcd(2, (4 % 2) = 0) (#: 1)
        // => 2

```

The remainder operation is called a total of  $1 + 1 + 1 + 1 = 4$  times.

03 May 2023

### [Exercise 1.19](#)

**There is a clever algorithm for computing the Fibonacci numbers in a logarithmic number of steps.** Recall the transformation of the state variables  $a$ ,  $b$  in the `fib_iter` process of section [1.2.2](#):  $a \leftarrow a + b$ , and  $b \leftarrow a$ .

Call this transformation  $T$ , and observe that applying  $T$  over and over again  $n$  times, starting with 1 and 0, produces the pair  $Fib(n + 1)$  and  $Fib(n)$ .

**In other words, the Fibonacci numbers are produced by applying  $T^n$ , the  $n^{th}$  power of the transformation  $T$ , starting with the pair (1, 0).**

Now consider  $T$  to be the special case of  $p = 0$ ,  $q = 1$  in a family of transformations  $T_{pq}$ , where  $T_{pq}$  transforms the pair  $(a, b)$ , according to  $a \leftarrow bq + aq + ap$  and  $b \leftarrow aq + bp$ .

Show that if we apply such transformation,  $T_{pq}$ , **twice**, the effect is the same as using a **single** transformation  $T_{p'q'}$  of the same form where  $p', q'$  are computed in terms of  $p, q$ .

This gives us an explicit way to square these transformations, and thus we can compute  $T^n$  (logarithmically) using successive squaring, as in the `fast_expt` function.

Put all this together to complete the following function, which runs in a logarithmic number of steps. (This exercise was suggested by Joe Stoy, based on an example in Kaldewaij 1990.)

```
function fib(n) {
    return fib_iter(1, 0, 0, 1, n);
}
function fib_iter(a, b, p, q, count) {
    return count == 0
        ? b
        : is_even(count)
        ? fib_iter(a,
                    b,
                    <<??>>,          // compute p'
                    <<??>>,          // compute q'
                    count / 2)
        : fib_iter(b * q + a * q + a * p,
                    b * p + a * q,
                    p,
                    q,
                    count - 1);
}
```

### Solution.

This was challenging in a rather different manner. I took a lot of time to solve this one. A great difficulty was conceptual.

How does one even *begin* to think of something like this?

Consider a step-by-step evolution of the implementation of  $Fib(n)$ : Given an integer  $n \geq 0$ , find the  $n^{th}$  Fibonacci number:

- 1) A rudimentary recursive formulation comes to mind almost immediately:  
 $Fib(n) = Fib(n - 1) + Fib(n - 2)$  with an appropriate base case. Time complexity:  
 $O(\phi^n)$  and space complexity:  $O(1)$ .

- 2) That implementation is a straightforward translation of the function definition into a working program in any programming language that supports recursion. Application of this function to an integer argument  $n > 1$  causes recursive applications of the same function to smaller arguments and as we saw in [tree recursion](#) above, without remembering the results of computations, a lot of redundancy ensues. We calculate  $Fib(m)$ ;  $m < n$  many times without realizing it. If we think of  $Fib(0)$  or  $Fib(1)$  as one atomic step (because only when the argument is either 0 or 1 does the base case occur), then we need to find out how many times we end up calling the function  $Fib$  with argument 0 or 1. As it turns out, this number is exponential in  $\phi$ , the golden ratio whose value is about 1.618. **If  $Fib(n)$  takes  $k$  steps, then  $Fib(n + 1)$  takes  $1.618k$  steps!**

So, we think of memoization, that is, remembering the previous results. This cuts the number of steps because we never recalculate the result that we have (previously) calculated and stored. Clearly, we need some memory to store the results so that we can recall and reuse them over and over. We trade space for the number of steps required to complete the task. Time complexity =  $O(n)$  since we need to calculate every Fibonacci number (less than the  $n^{th}$ ) once. Space complexity =  $O(n)$ .

- 3) A [variant](#) that utilizes the transformation this problem refers to in that it uses two more arguments,  $a$ ,  $b$ , and applies the transformation  $[a \leftarrow a + b, b \leftarrow a]$   $n$  times to get  $Fib(n)$ . This is a beautiful transformation and since the tail calls can be optimized, stack space can be reused. Since each transformation finds a new Fibonacci number in  $b$ , the  $n^{th}$  Fibonacci number is found in applying the transformation  $n$  times. Time complexity:  $O(n)$ . Space complexity:  $O(1)$ .

What makes this [implementation](#) (`function fib(a, b, n)`) beautiful? Perhaps the mysteriousness. There is a mystery involved in thinking of providing additional arguments. It is not immediately clear if such an implementation can be arrived at just by thinking. Perhaps intuition is needed. If we are told that someone arrived at this implementation as their first attempt after looking at the recursive  $Fib$  function definition (by cases) what would have gone in their mind? Perhaps a lot of guesswork was involved (like George Polya would have said).

**Is this the best?** As a student of computer science, this question constantly haunts us.

How does one now suspect that a better algorithm exists? From our understanding of orders of growth and functions governing commonly found natural processes, we know that the log function grows much less rapidly than the identity function. But that doesn't mean that this problem can be solved in the number of steps that is logarithmic in  $n$ .

There is one observation that gives one some hint that perhaps an  $O(\log n)$  algorithm for  $Fib(n)$  exists.



See [Exercise 1.13](#) where we proved that  $Fib(n) = nint(\phi^n)$ . Since we learned that “raising a real number to an integral power” is a problem that can be solved in  $O(\log n)$  steps ([Exercise 1.16](#) and constant space, because tail calls can be optimized), it follows that, at least in math, finding  $Fib(n)$  is as difficult as raising  $\phi$  to  $n$ , for which a logarithmic algorithm (with constant space) is known to exist.

But this problem (conceived by Joe Stoy, a professor of Computer Science, after referring to a book, *Programming: The Derivation of Algorithms*, by Anne Kaldewaij) takes us on a higher level of abstraction. How does one think of the transformation  $[a \leftarrow a + b, b \leftarrow a]$  to be a *special case* of a more general transformation  $T: [a \leftarrow aq + bq + ap, b \leftarrow aq + bp]$  with  $[p = 0, q = 1]$ ? Is such thinking rooted in the training of mathematicians in “generalizing results”? Perhaps it is so.

After quite some contemplation, I was able to use the transformation  $T$  twice in succession on the two variables  $a, b$ . I don’t remember how this thought came into my mind, but somehow I thought that applying  $T$  twice is same as applying  $T^2$  once and some scribbling with paper and pencil helped. What follows is a fair version of a lot of rough scratchwork:

Inputs of T [ $(p, q)$ are constant]	Outputs of T
$a$ $b$	$a \leftarrow (a + b)q + ap$ $b \leftarrow aq + bp$
$(a + b)q + ap$ $aq + bp$	$a \leftarrow ((a + b)q + ap + aq + bp)q + ((a + b)q + ap)p$ $b \leftarrow ((a + b)q + ap)q + (aq + bp)p$  On simplification, $a \leftarrow a(2q^2 + 2pq + p^2) + b(q^2 + 2pq)$ $b \leftarrow a(q^2 + 2pq) + b(q^2 + p^2)$  Now, if $p' = p^2 + q^2$ and $q' = q^2 + 2pq$ , then the above is analogous to: $a \leftarrow a(p' + q') + bq'$ $b \leftarrow aq' + bp'$  We call this transformation $T_{p'q'}$ since its form is the same as that of $T_{pq}$ .

Thus, the effect of applying  $T_{pq}$  twice is equivalent to a single application of  $T_{p'q'}$ . This also means that applying  $n$  times  $T_{pq}$  is equivalent to applying  $\frac{n}{2}$  times  $T_{p'q'}$ . So, the idea seems to emerge: We apply  $T_{p'q'}$  to cut the number of steps in half (when  $n$  is even) and  $T_{pq}$  to reduce it by 1 (when  $n$  is odd). This results in a logarithmic time algorithm!

The code now becomes:

```

        square(p) + square(q),          // compute p'
        square(q) + double(p * q),    // compute q'

function fib_log(n) {
  function fib_iter(a, b, p, q, count) {
    return count === 0
      ? b
      : is_even(count)
        ? fib_iter(a,
                    b,
                    square(p) + square(q),          // compute p'
                    square(q) + double(p * q),    // compute q'
                    count / 2)
        : fib_iter(b * q + a * q + a * p,
                    b * p + a * q,
                    p,
                    q,
                    count - 1);
  }
  return fib_iter(1, 0, 0, 1, n);
}
fib_log(100); // => 354224848179261900000

```

However, there is one more observation: The smallest value of  $n$  for which [fib\\_log](#) and [fib1](#) differ on my machine is 80. This is mostly due to the integer overflow. I should implement a BigInteger solution in Java and see exactly what the 80th Fibonacci number is. **TODO**.

04 April 2023

### 1.2.4 Fast Exponentiation

A straightforward **recursive** implementation for the (recursive) process that raises a number  $b$  to an integral power  $n$  follows:

```
function expt(b, n) {
  return n === 0
    ? 1
    : b * expt(b, n - 1);
}
```

An equivalent iterative process can be quickly realized. But since the tail call can be optimized, it saves space:

```
function expt(b, n) {
  function expt_iter(b, counter, product) {
    return counter === 0
      ? product
      : expt_iter(b, counter - 1, b * product);
  }
  return expt_iter(b, n, 1);
}
```

A faster algorithm that uses a recursive process is also pretty well-known. The book provides the following version:

```
function square(x) {
  return x * x;
}
```

```
function is_even(n) {
  return n % 2 === 0;
}
```

```
// faster exponentiation - uses recursive process (deferred ops)
function fast_expt(b, n) {
  return n === 0
    ? 1
    : is_even(n)
      ? square(fast_expt(b, n / 2))
      : b * fast_expt(b, n - 1);
}
```

I have a slightly improved version (that uses less memory) of this:

```
// improved exponentiation v2 - uses recursive process (deferred ops)
function fast_expt2(b, n) {
  return n === 0
    ? 1
    : is_even(n)
      ? square(fast_expt(b, n / 2))
      : b * square(fast_expt(b, (n - 1) / 2));
}
```

Whereas the `expt` function is  $O(n)$ , `fast_expt` is  $O(\log n)$  in terms of number of steps.

[The next exercise \(1.16\)](#) is a continuation of the above discussion. In general, SICP (this chapter in particular) is teaching us to think in terms of the computational processes functions generate. Remember, we have no other device, but functions, to implement those processes. The two computational processes of interest are:

- 1) Recursive
- 2) Iterative

Once we conceive and implement a recursive process by recursive functions, we start thinking of how to implement its iterative equivalent. As of now, I guess that the **type of functions** dictates whether we can convert a recursive process into its iterative equivalent. Perhaps this goes into the theory of recursive functions.

### [Exercise 1.16](#)

Design a function that evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps, as does `fast_expt`.

(Hint: Using the observation that  $(b^{n/2})^2 = (b^2)^{n/2}$ , keep, along with the exponent  $n$  and the base  $b$ , an additional state variable  $a$ , and define the state transformation in such a way that the product  $ab^n$  is unchanged from state to state. At the beginning of the process  $a$  is taken to be 1, and the answer is given by the value of  $a$  at the end of the process. In general, the **technique of defining an invariant quantity that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.**)

### Solution.

This is surprisingly difficult. At least for me. I have spent quite some time on this one. I guess I understand using  $(b^{n/2})^2 = (b^2)^{n/2}$ , but I don't yet get how to actually do it and I don't want to see their solution. But I do think this is related to the idea of function composition.

I want to exploit the idea of the accumulator (the argument `acc`) that holds the result of an “ongoing calculation”, but the boundary conditions are tricky. I tried an experiment shown in the table below with an exponent that is a power of 2:

base	exponent	acc	notes
------	----------	-----	-------

$b$	8	1	It seems clear that <code>acc</code> is returned when <code>exponent</code> becomes 0. What if we have: $acc = acc \cdot b^{oldexp/newexp}$ ?
$b$	4	$1 \cdot b^{8/4} = b^2$	
$b$	2	$b^2 \cdot b^{4/2} = b^4$	
$b$	1	$b^4 \cdot b^{2/1} = b^6$	Fails here. We want this to be $b^8$ .

We want to retain the  $\Theta(\log n)$  running time and yet have an iterative process. How should `acc` be manipulated?

- Its initial value should be 1 because that takes care of the case that `exponent` is 0.
- When `exponent` is even, we want the function call `square(fast_expt(b, n / 2))` (with the introduction of an additional argument like `acc`) to result in a call to something like `fast_expt(b, n/2, square(...))`. This is because our halving of `exponent` takes us toward  $\Theta(\log n)$  number of steps, and the third argument, `acc`, helps preserve a partial result. What should `acc` be? If we simply did `fast_expt(b, n/2, square(acc))`, we can achieve  $\Theta(\log n)$  runtime complexity, but we can't increase `acc` from its initial value of 1 (repeatedly squaring 1 results in 1).
- Should we preserve the powers of the base and accumulator separately? In keeping with  $(b^{n/2})^2 = (b^2)^{n/2}$ , every time the `exponent` is halved, we can square the base:  

```
function fast_expt(base, exponent, powers, acc) {
    return exponent == 0
        ? acc
        : is_even(exponent)
            ? fast_expt(base, exponent / 2, square(powers), acc *
square(powers))
            : fast_expt(base, (exponent - 1) / 2, square(powers),
acc * base * square(powers));
}
```
- The above does not work.

I need to revisit. **TODO**.

### [Exercise 1.17](#)

The exponentiation algorithms in this section are based on performing exponentiation by means of repeated multiplication. In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication function (in which **it is assumed that our language can only add, not multiply**) is analogous to the `expt` function:

```
function times(a, b) {
    return b === 0
        ? 0
        : a + times(a, b - 1);
}
```

This algorithm takes a number of steps that is linear in  $b$ .

Now suppose we include, together with addition, the functions `double`, which doubles an integer, and `halve`, which divides an (even) integer by 2. **Using these, design a multiplication function analogous to `fast_expt` that uses a logarithmic number of steps.**

**Solution.**

Before we make progress on the solution, let's follow a good convention that is used in arithmetic: In  $c = a \times b$ ,  $a$  is the *multiplier* (the first operator), and  $b$  is the *multiplicand* (the second operator). Although  $a \times b = b \times a$  (that is, they are the same as far as the *result* is concerned), we should have a convention about what is what. In what follows, we say *multiplier times multiplicand*.

The given function, `times`, is  $\Theta(n)$  and also it represents a recursive process since there are deferred operations (addition) after the recursive function call returns. We are asked to improve the running time to  $\Theta(\log n)$ . Let's tackle that requirement first.

```
function double(a) {
    return a * 2;
}

function halve(a) { // performs int div; call on an even number
    return a / 2;
}

function fast_times(b, a) {
    return b === 0
        ? 0
        : is_even(b)
          ? double(fast_times(halve(b), a))
          : a + double(fast_times(halve(b - 1), a));
}

fast_times(99, 9); // => 891
```

At every step, we halve the second argument. Therefore, the number of steps is  $\Theta(\log n)$ . But this version is not tail-recursive, since there are outstanding operations after the previous call returns.

### Exercise 1.18

Using the results of exercises [1.16](#) and [1.17](#), devise a function that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps. ([See the footnote.](#))

#### **Solution.**

This remains a difficult problem as of now (04/19/2023 6.42 pm). I have spent a lot of time on this one. Is it the time to peek at the solution in the book?

I am tempted to see. But no, I will hang on.

The [fast\\_times](#) function above needs  $\Theta(\log n)$  steps. But it also needs  $\Theta(\log n)$  space because there are deferred operations: function calls or arithmetic operations. I reproduce that function here:

```
function fast_times(b, a) {
  return b === 0
    ? 0
    : is_even(b)
      ? double(fast_times(halve(b), a))
      : a + double(fast_times(halve(b - 1), a));
}
// the initial call
fast_times(99, 9); // => 891
```

I clearly see (intuit) that there should be some accumulator that captures the result of in-flight calls. Here is an attempt:

```
// b x a: b is the multiplier which we repeatedly halve
function fast_times_iter_attempt_1(b, a, acc) { //acc=0 to begin with
  return b === 0
    ? acc
    : is_even(b)
      ? fast_times_iter_attempt_1(halve(b), a, acc + double(a))
      : fast_times_iter_attempt_1(halve(b - 1), a, acc + a +
double(a));
}

fast_times_iter_attempt_1(1, 4, 0); // => ?
```

```

Call tree:
fast_times_iter_attempt_1(1, 4, 0)
- fast_times_iter_attempt_1(0, 4, 12) // 0 + 4 + double(4) = 12

// => 12

```

This is clearly wrong. We are doing more than necessary. What is the invariant in this case?

$$2n \times a = \frac{2n}{2} \times a + n \times a$$

$$(2n + 1) \times a = \frac{2n}{2} \times a + \left(\frac{2n}{2} + 1\right) \times a$$

How to weave this in terms of acc? How about

$$2n \times a + acc = \frac{2n}{2} \times a + n \times a + acc$$

$$(2n + 1) \times a + acc = \frac{2n}{2} \times a + \left(\frac{2n}{2} + 1\right) \times a + acc$$

This is tricky. It's also because we are trying to optimize too much. What if when the multiplier is odd, we just decrease it by 1 (so that on the next call the multiplier is even and we can halve it)?

```

// b x a: b is the multiplier which we repeatedly halve
function fast_times_iter_attempt_2(b, a, acc) { //acc=0 to begin with
  return b === 0
    ? acc
    : is_even(b)
      ? fast_times_iter_attempt_2(halve(b), a, acc + double(a))
      : fast_times_iter_attempt_2(b - 1, a, acc + a);
}
// fast_times_iter_attempt_2(4, 7, 0) // =>

```

I think time has come for me to consult Apoorv who has solved this problem (apparently he got really angry one day and he was able to use his energy to solve it). Update on 03 May 2023.

**TODO.**

## 28 March 2023

### 1.2.3 Orders of Growth

We say that the resources,  $R(n)$ , used by a computational process are  $\Theta(n)$  when, for some positive constants  $k_1$ ,  $k_2$ ,  $n_0$  and some function  $f(n)$ , we have

$$k_1 f(n) \leq R(n) \leq k_2 f(n) \quad \forall n > n_0$$



### Exercise 1.14

Draw the tree illustrating the process generated by the `count_change` function of section [1.2.2](#) in making change for 11 cents. What are the orders of growth of the space and number of steps used by this process as the amount to be changed increases?

**Solution.**

```
change(11)
- change_using(11, 1) <- [1]
  - change_using(-39, 1) <- 0
- change_using(11, 2) <- [1]
  - change_using(-14, 2) <- 0
- change_using(11, 3) <- [1]
  - change_using(1, 3) <- 1 [1]
    - change_using(-9, 3) <- 0
  - change_using(1, 4) <- [1]
    - change_using(-4, 4) <- 0
    - change_using(1, 5) <- [1]
      - change_using(0, 5) <- 1
      - change_using(1, 6) <- 0
- change_using(11, 4)
  - change_using(6, 4)
    - change_using(1, 4) <- [1]
      - change_using(-4, 4) <- 0
      - change_using(1, 5) <- [1]
        - change_using(0, 5) <- 1
        - change_using(1, 6) <- 0
    - change_using(6, 5)
      - change_using(5, 5)
        - change_using(4, 5)
          - change_using(3, 5)
            - change_using(4, 6) <- 0
            - change_using(5, 6) <- 0
          - change_using(6, 6) <- 0
      - change_using(11, 5)
```

. . .

Instead of completing this task by hand, I modified the code to count the number of calls to the function `change_using` made by the original call: `change(11)`. That number is 55. For `change(100)`, `change_using` is called 15499 times. It appears exponential. We've got to prove it though. An accurate analysis is challenging.

The space requirements vary as the depth of the function call tree and the time requirements as the number of nodes in it. Let's first take up the number of steps – that number relates to the

time requirements. Let  $T(a, 5)$  denote the number of steps required to change the amount  $a$  using the 5 coin denominations of our choice. The coins are arranged in order – from the numerically highest denomination,  $d_1$  through the lowest,  $d_5$ . Let's also define five more integers –  $m_1$  through  $m_5$  – such that:

$$m_1 d_1 \leq a < (m_1 + 1) d_1$$

$$m_2 d_1 \leq a < (m_2 + 1) d_2$$

$$m_3 d_1 \leq a < (m_3 + 1) d_3$$

$$m_4 d_1 \leq a < (m_4 + 1) d_4$$

$$m_5 d_1 \leq a < (m_5 + 1) d_5$$

First call to `change_using(a, 1)` will result in

- 1) one call to `change_using(a-d1, 1)`, and
- 2) one call to `change_using(a, 2)`.

How many additional function calls are made with the second argument equal to 1? Either  $m_1+1$  calls or  $m_1+2$  calls: every subsequent call to `change_using` will reduce the amount by  $d_1$ . This goes on  $m_1+1$  times if  $m_1 \% d_1 == 0$ , and  $m_1+2$  times otherwise.

This is becoming rather too complicated.

### [Exercise 1.15](#)

The sine of an angle (specified in radians) can be computed by making use of the following function:

$$\sin(x) = \begin{cases} x & \text{if } x \leq 0.1^e, \\ 3\sin\left(\frac{x}{3}\right) - 4\sin^3\left(\frac{x}{3}\right) & \text{otherwise} \end{cases}$$

It can be incorporated in the following functions:

```
function cube(x) {
    return x * x * x;
}
function p(x) {
    return 3 * x - 4 * cube(x);
}
function sine(angle) {
    return ! (abs(angle) > 0.1)
        ? angle
        : p(sine(angle / 3));
}
```

1. How many times is the function `p` applied when `sine(12.15)` is evaluated?

2. What is the order of growth in space and number of steps (as a function of  $a$ ) used by the process generated by the `sine` function when `sine(a)` is evaluated?

**Solution.**

1) How many times is the function `p` applied when `sine(12.15)` is evaluated?

These functions are defined in Source-1 in a very peculiar (or should we say functional) style!

My first reaction was to define the `sine` function thus:

```
function my_sine(angle) {  
  return abs(angle) <= 0.1  
    ? angle  
    : 3 * my_sine(angle / 3) - 4 * my_sine(angle / 3) *  
      my_sine(angle / 3) * my_sine(angle / 3);  
}
```

This definition just came naturally to me.

Here is another variant of the same:

```
function my_sine2(angle) {  
  return abs(angle) <= 0.1  
    ? angle  
    : 3 * my_sine2(angle / 3) - 4 * cube(my_sine2(angle / 3));  
}
```

The definition of `my_sine` represents a recursive process. Clearly, there are deferred operations after the recursive calls return. As you can see, my definition does not require the function `p` at all!

On the other hand, their definition uses the **function composition**. Consider the following functions along with the  $\sin(x)$  function as defined by cases above:

$$p(x) = 3x - 4x^3$$

$$\text{cube}(x) = x^3$$

Therefore,

$$p(\sin(x/3)) = 3 \sin(x/3) - 4 \text{cube}(\sin(x/3))$$

This gives the following redefinition of  $\sin(x)$ :

$$\sin(x) = \begin{cases} x & \text{if } |x| \leq 0.1^c, \\ p(\sin(\frac{x}{3})) & \text{otherwise} \end{cases}$$

Is this simpler (or more elegant) than `my_sine`? And what about efficiency?

The new definition of  $\sin(x)$  translates rather simply to the following set of functions:

```
function cube(x) {
  return x * x * x;
}

function p(x) {
  return 3 * x - 4 * cube(x);
}

function sine(x) {
  return abs(x) <= 0.1
    ? x
    : p(sine(x/3));
}
```

The substitution model tells us what the function, `sine(x)`, call tree looks like when `x = 12.15`:

```
sine(12.15)
- p(sine(4.05))
  - p(sine(1.35))
    - p(sine(0.45))
      - p(sine(0.15))
        - p(sine(0.05))
```

There are 5 calls to `p` in this case.

There is another way to look at it. Each call to `sine(x)` results in one call to `sine(x/3)` unless `x` is too small, in which case it simply returns `x`. Consider that one function application, `sine(x)`, results in applying `n` times the function `p`:

```
sine(x)
- p(sine(x/3))
- p(p(sine(x/32)))
- p(p(p ... p(sine(x/3n))) (this is the nth application of p)
```

The last argument must return itself:  $\text{ sine}(x/3^n) = x/3^n$ . For this equality,  $x/3^n$  must be less than or equal to 0.1.

$$\text{Thus, } \frac{x}{3^n} \leq 0.1 \rightarrow 10x \leq 3^n \rightarrow 3^n \geq 10x$$

$$\rightarrow n \geq \log_3(10x).$$

Thus, the number of times `p` is called for an argument `x` is roughly <sup>2</sup>

$$\log_3(10x) = O(\log_3 x) = O(\log_2 x)$$

2) What is the order of growth in space and number of steps (as a function of  $a$ ) used by the process generated by the `sine` function when `sine(a)` is evaluated?

**Solution.**

The number of steps in the evaluation of `sin(a)` is equal to the number of times the function `p` is called. This number, as shown above, is  $O(\log_2 a) = k \cdot \log_2 a$ .

As the function call tree shows, it is a recursive process of recursive function calls because there are deferred operations. After a previous recursive function call returns, there is still some work to do (application of function `p`) on the returned value before the current function call can return. Thus, every step will be in flight until the last function call application returns for a sufficiently small value of its argument. Therefore the space requirements are also  $O(\log_2 a)$ .

How about the `my_sine` implementation?

As coded above (that is without any memoization or because of redundant computations),

`my_sine` is rather inefficient: one `sin(a)` call results in 4 calls to `sin(a/3)`. Although the number of steps it takes is still  $O(\log n)$ , its *constant factors* are worse. This results in a slower execution. Memoization can of course speed up its execution (e.g. we can cache the results of a previous call in a local variable or we can employ a function like `cube(x)`).

Is it possible to use in this case the accumulator trick (using additional arguments) like what we used in the case of the factorial function or the fibonacci function? It appears difficult.

Are the two recursive functions `sin(x)` (as defined [originally](#)) and `factorial(x)` somehow *different*? If yes, how?

---

<sup>2</sup>  $f$  and  $g$  are two different functions and when  $f = O(g)$ , the meaning of the  $=$  sign is subtly different from its usual meaning.

9-27 March 2023

### A Note on Memoization or Tabulation

To avoid redundant calculations, memoization (caching) of the computations helps. This assumes that the lookup is much faster than the recalculation. Thus, if a lookup can fetch in  $O(1)$  the value of `fib(5)` that we have already calculated, we save a lot of time and do not perform any redundant computations. In Source-1 so far though, we don't know how to implement a table lookup. In an imperative language, we use a data structure called a hash table.

### [Exercise 1.11](#)

A function  $f$  is defined by the rules

$$f(n) = \begin{cases} n, & \text{if } n < 3 \\ f(n-1) + 2f(n-2) + 3f(n-3) & \text{otherwise} \end{cases}$$

Write a JavaScript function that computes  $f$  by means of a recursive process. Write a function that computes  $f$  by means of an iterative process.

#### **Solution.**

Quite a rapidly growing function!

The recursive function that represents a recursive process:

```
function f111_recursive(n) {  
    return n < 3  
        ? n  
        : f111_recursive(n - 1) + 2 * f111_recursive(n - 2) + 3 *  
          f111_recursive(n - 3);  
}
```

The recursive function that represents a iterative process:

```
function f111(n) {  
    function f111_iter(a, b, c, ctr) {  
        return ctr === 2  
            ? c  
            : f111_iter(b, c, a + 2 * b + 3 * c, ctr + 1);  
    }  
    return f111_iter(1, 1, 1, 0);  
}
```

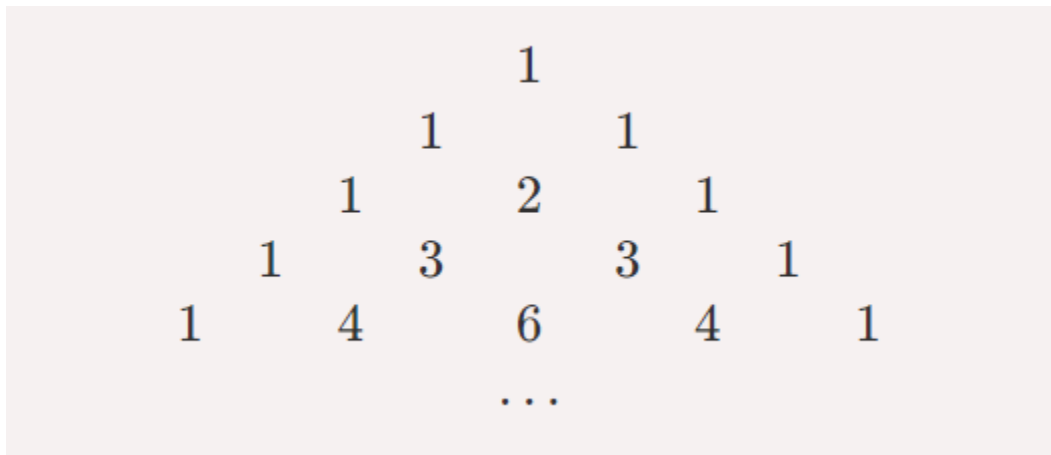
```

        : ctr === 1
        ? b
        : ctr === 0
        ? a
        : f111_iter(b, c, c + 2 * b + 3 * a, ctr - 1);
    }
    return f111_iter(0, 1, 2, n);
}

```

### [Exercise 1.12](#)

The following pattern of numbers is called *Pascal's triangle*.



The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it. Write a function that computes elements of Pascal's triangle by means of a recursive process.

#### **Solution.**

Knowing a bit of algebra helps. The rows of Pascal's triangle represent the coefficients of the terms in the expansion of the binomial  $(x + y)^n$ ,  $n \geq 0$ .

```

// n is the row (>= 0), and c column (0 <= c <= n)
function pascal(n, c) {
    return n === 0 || c === 0 || c === n
        ? 1
        : pascal(n - 1, c - 1) + pascal(n - 1, c);
}

```

Can this be realized using an iterative process? (Of course, there are other ways of doing this iteratively because  $pascal(n, c) = P(n, c) = \frac{n!}{(n-c)!}$ . And we can easily compute

the ratio of two factorials in  $O(n)$  time (and constant space), but we want to use this given recursive function that represents an iterative process.)

Perhaps memoization will help?

**TODO.**

### Exercise 1.13

Prove that  $\text{Fib}(n)$  is the closest integer (*nint*) to  $\varphi^n / \sqrt{5}$  where  $\varphi = \frac{1+\sqrt{5}}{2}$ .

Hint: Use mathematical induction and the definition of the Fibonacci numbers to prove that

$$\text{Fib}(n) = \frac{\varphi^n - \psi^n}{\sqrt{5}}, \text{ where } \psi = \frac{1-\sqrt{5}}{2}.$$

**Solution.**

**Proof-1.** We use mathematical induction as suggested.

**Base case:** Consider  $n = 0$ .  $\text{Fib}(0) = \frac{\varphi^0 - \psi^0}{\sqrt{5}} = \frac{1-1}{\sqrt{5}} = 0$ . This is true.

**Inductive Hypothesis:** We assume  $\text{Fib}(r)$  to be true for some integer  $r$  and prove  $\text{Fib}(r+1)$ .

$$\therefore \text{Fib}(r) = \frac{\varphi^r - \psi^r}{\sqrt{5}} \text{ ----- (1.13.1)}$$

Similarly,

$$\text{Fib}(r-1) = \frac{\varphi^{r-1} - \psi^{r-1}}{\sqrt{5}} \text{ ----- (1.13.2)}$$

Then, adding the LHS's of 1.13.1 and 1.13.2, and applying the definition of Fibonacci numbers, we get  $\text{Fib}(r+1)$ .

$$\begin{aligned} \therefore \text{Fib}(r+1) &= \text{Fib}(r) + \text{Fib}(r-1) = \frac{\varphi^r - \psi^r}{\sqrt{5}} + \frac{\varphi^{r-1} - \psi^{r-1}}{\sqrt{5}} \\ &= \frac{1}{\sqrt{5}} ((\varphi^r + \varphi^{r-1}) - (\psi^r + \psi^{r-1})) = \frac{1}{\sqrt{5}} (\varphi^{r+1}(\frac{1}{\varphi} + \frac{1}{\varphi^2}) - \psi^{r+1}(\frac{1}{\psi} + \frac{1}{\psi^2})) \\ &= \frac{1}{\sqrt{5}} (\varphi^{r+1} - \psi^{r+1}) \end{aligned}$$

(Since  $\varphi$  and  $\psi$  are peculiar numbers and satisfy the equations  $\varphi^2 = \varphi + 1$  and  $\psi^2 = \psi + 1$ , both  $\frac{1}{\varphi} + \frac{1}{\varphi^2}$  and  $\frac{1}{\psi} + \frac{1}{\psi^2}$  equal 1).

**I**



So, we proved  $Fib(n) = \frac{\varphi^n - \psi^n}{\sqrt{5}}$ . But that does not mean that  $Fib(n) = nint(\frac{\varphi^n}{\sqrt{5}})$ .

(The  $nint(x)$ ,  $x \in \mathbb{R}$  function is the function that returns the integer nearest to  $x$ .)

We can say from above that

$$Fib(n) = nint(\frac{\varphi^n}{\sqrt{5}}) \leftrightarrow nint(\frac{\varphi^n}{\sqrt{5}}) = \frac{\varphi^n - \psi^n}{\sqrt{5}}$$

Thus, we want to prove that  $\frac{\varphi^n - \psi^n}{\sqrt{5}}$  is the integer nearest to  $\frac{\varphi^n}{\sqrt{5}}$ . Let's observe how  $\frac{\psi^n}{\sqrt{5}}$  (and

$\psi \approx -0.618034$ ) changes with  $n$ :

n	0	1	2	3	4	5	6
$\frac{\psi^n}{\sqrt{5}}$	0.4472	-0.2764	0.1708	-0.1055	0.065	-0.0404	0.025

Since  $\frac{\varphi^n - \psi^n}{\sqrt{5}}$  is the exact value of  $Fib(n)$  (which is an integer by definition) and that  $|\frac{\psi^n}{\sqrt{5}}|$

rapidly approaches 0, it seems that  $\frac{\varphi^n - \psi^n}{\sqrt{5}} \approx \frac{\varphi^n}{\sqrt{5}}$  and  $Fib(n) = nint(\frac{\varphi^n}{\sqrt{5}})$ .

(The proof is rather incomplete.)

08 March 2023

### 1.2.2 Tree Recursion

Another pattern of recursion is when a call to a function with an argument (input size) results in multiple calls to the same function albeit all with reduced argument values. This is akin to saying that we break a problem into multiple problems of smaller arguments. This makes linear recursion a special case of tree recursion. In linear recursion, for instance, a call to a function  $f$  with argument  $n$ ,  $f(n)$ , may result in a recursive call  $f(n-1)$ , which in turn results in another call,  $f(n-2)$ , and so on, until [as we say] the recursion bottoms out when the function call simply returns (something). In such linear recursion, a function call  $f(n)$  results roughly into  $n$  calls to  $f$  and that's "linear" recursion.

Now consider that [a call to]  $f(n)$  results in two calls to  $f(n-1)$  unless  $n$  is 0 in which case it simply returns 1. The shape of the function calls is similar to a tree (upside down, with root at

the top and leaves at the bottom). This is why it is called tree recursion. This function models any process that terminates in  $2^n$  steps.

A similar construct is the familiar Fibonacci sequence. It is defined as the sequence of natural numbers generated by the following function upon applying the argument that denotes the ordinality:

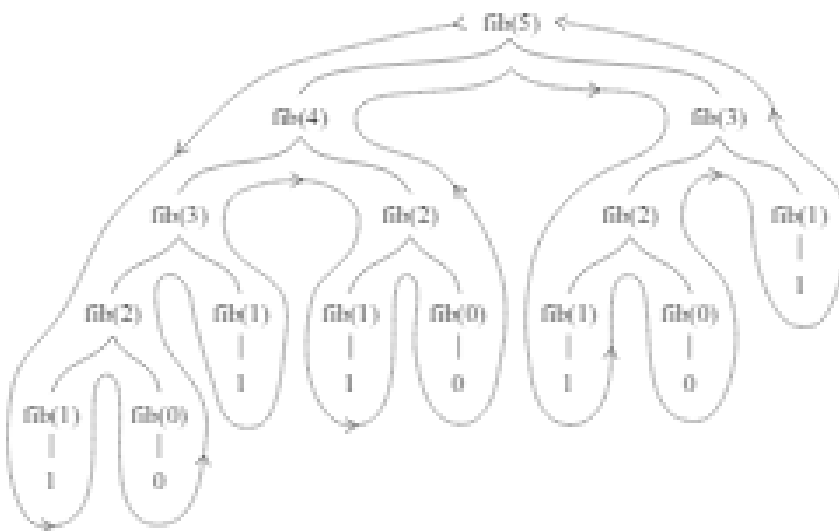
$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{Fib}(n - 1) + \text{Fib}(n - 2) & \text{otherwise} \end{cases}$$

The Fibonacci sequence is: 0, 1, 1, 2, 3, 5, 8, ...

A straightforward implementation of the above function (named, say, `fib`) accepts an integer.

```
function fib(n) {
  return n === 0
    ? 0
    : n === 1
    ? 1
    : fib(n - 1) + fib(n - 2);
}
```

An invocation like `fib(5)` results in the following calls:



That resembles a tree.

A function call like `fib(5)` results in a lot of redundant computation. We can work on optimizing the work, but the canonical function clearly illustrates a common pattern.

How many leaf nodes (each corresponding to either `fib(1)` or `fib(0)`) are there in a tree of `fib(5)`? Note that 1 and 0 are the only two arguments when the application of the function `fib` returns a value immediately: `fib(1) = 1`, `fib(0) = 0`.

This is a great question. I had never realized this before. We can use mathematical induction to attempt to find this. But before that, we can work on finding a pattern.

n	Fib(n)	Leaf(n) = Number of leaf nodes in the tree of fib(n)	Nonleaf(n) = Number of non leaf nodes in the tree of fib(n)
0	0	1 (counting the root as leaf)	0 (counting the root as leaf)
1	1	1 (counting the root as leaf)	0 (counting the root as leaf)
2	1	2 (1 fib(0), 1 fib(1))	1
3	2	3	2
4	3	5	4
5	5	8	7
6	8	13	12
...		...	
n	Fib(n)	Leaf(n-1) + Leaf(n-2) ??	Leaf(n) - 1 or NonLeaf(n-1) + NonLeaf(n-2) + 1 ??

Is this the pattern, giving the following (probable) definition of the function, `Leaf(n)`?

$$Leaf(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ Leaf(n-1) + Leaf(n-2) & \text{otherwise} \end{cases}$$

This is right because for any integer argument  $n \geq 2$ , in the function call tree of `fib(n)`, we have a left subtree rooted at `fib(n-1)`, and the right subtree rooted at `fib(n-2)`. Clearly, the number of leaf nodes in the left subtree is  $Leaf(n-1)$  and the number of leaf nodes in the right subtree is  $Leaf(n-2)$ . Any path from the root of the tree of  $Leaf(n)$ ;  $n \geq 2$  to a leaf passes through either the node representing `fib(n-1)` or the node representing `fib(n-2)`.

And how do we relate the two mathematical functions: *Leaf*, *Fib*?

Let's start with the obvious replacement:

$Leaf(n) = 1 = Fib(n)$ , when  $n = 1$  and

$Leaf(n) = 1 = 1 + Fib(n)$ , when  $n = 0$ .

$$Leaf(n) = \begin{cases} 1 + Fib(n), & \text{if } n = 0 \\ Fib(n), & \text{if } n = 1 \\ Leaf(n-1) + Leaf(n-2) & \text{otherwise} \end{cases}$$

If we observe the two sequences:

`Fib(n)`: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

`Leaf(n)`: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

we can conjecture that  $Leaf(n) = Fib(n+1)$ .

And we can use mathematical induction to prove that.

<TBD>

This function (`fib`) is instructive as a prototypical tree recursion, but it is a terrible way to compute Fibonacci numbers because it does so much redundant computation. As the `Leaf` function shows above, the number of times `fib(1)` or `fib(0)` will be called on a call to `fib(n)` is exponential in  $n$ . As above, this number of  $Fib(n+1)$ . A detailed analysis of this shows that Fibonacci numbers are intricately related to golden ratio,  $\phi \approx 1.618$  and that  $Fib(n) = \phi^n$  and this means an additive increase in  $n$  increases  $Fib(n)$  by a multiplicative factor – such growth is tremendous!

There are two fundamental aspects of this tree recursion (let's consider the [straightforward fib implementation above](#)):

- 1) Number of times the naive function gets called – this is the same as the total number of is clearly exponential in  $n$ .
- 2) The memory requirements of the implementation – is this exponential in  $n$ ? No. It is not. At any point in time, the biggest number of function calls that have not returned yet is proportional to the “depth of the Fibonacci tree”. The depth of the tree is logarithmic in  $n$ .

Can the above implementation that describes a recursive **process** be morphed to describe an iterative **process** to find `Fib(n)` for a given  $n$ ?

Fortunately we can do that!

```
function fib(n) {
  function fib_iter(a, b, c) {
    return c === 0
      ? b
      : fib_iter(a + b, a, c - 1);
  }
  return fib_iter(1, 0, n);
}
```

This trick results in the following calls for `fib(5)`:

```
- fib(5)
  - fib_iter(1, 0, 5)
    - fib_iter(1, 1, 4)
      - fib_iter(2, 1, 3)
        - fib_iter(3, 2, 2)
          - fib_iter(5, 3, 1)
            - fib_iter(8, 5, 0)
```

This returns the value 5. And since there are no deferred operations, the space needed is  $O(1)$ : A tail-recursive compiler implementation should<sup>3</sup> reuse a stack frame. In other words, the call `fib_iter(5, 3, 1)` is identical to the call `fib_iter(3, 2, 2)`. The compiler should ideally have no knowledge that one of these function calls resulted in the other function call (an evidence that tail-calls are optimized).

We were lucky to find an implementation trick that converted a tree recursion into a linear recursion (iteration). Are we always going to be so lucky?

### 1.2.2.1 The Coin-change Problem

Consider that we have an infinite supply of coins of denomination ‘ $a$ ’. We are given an amount ‘ $n$ ’. Will we always be able to change the amount using those coins?

---

<sup>3</sup> [Here's what](#) the computer actually does with an executable generated by an optimizing C compiler

One might say – perhaps influenced by the division algorithm we all learn in grade school – that if ‘a’ is a multiple of ‘n’, we can, not otherwise.

That’s a fine answer, but we want to do it more slowly. We want to actually experience various cases rather obviously. We imagine running a “coin-change service” and a customer comes to our doorstep with an integer amount ‘n’ and asks for a change in coins of denomination ‘a’.

Here is my first attempt:

```
// returns 1 if n can be changed using coins of denomination a
function f1(n, a) {
    return n < a
        ? 0
        : n === a
        ? 1
        : f1(n - a, a);
}
```

The code seems straightforward.

**Apoorv found a bug in this implementation!** It revolves around this question: **How many ways are there to change 0 cents using the coins of denomination ‘a’?**

I argued that it is 0. But he insisted that it is 1. And he is right! Is this just an unimportant detail? No, boundary conditions are extremely important in computing. There is exactly one way of changing 0 cents using the coins of denomination ‘a’: You give 0 coins –  $0 = 0 \cdot a$ . This is not true for any other integer amount that is less than ‘a’.

On the other hand, if you have no coins of any denomination (e.g. you went bankrupt) then there is no way for you to change any amount (other than 0).

```
function f1(n, a) {
    return n === 0
        ? 1
        : n < a
        ? 0
        : f1(n-a, a);
}
```

This can be extended to doing the job with coins of two different denominations, ‘a’, and ‘b’:

```
function f2(n, b, a) {
    return n > b
        ? f2(n - b, b, a) + f1(n, a)
        : n === b
        ? 1
        : 0;
}
```

```

? 1 + f1(n, a)
: n > a
? f1(n - a, a)
: n === a
? 1
: 0;
}

```

A generalization of the above is what we seek. We have a given number of denominations (e.g. half dollars, quarters, dimes, nickels, and pennies). How do we go about doing that?

To hold the coins of the given denominations, in an imperative programming language, one would use a list or an array that provides indexed access. There is no such provision in our programming environment yet and we need to satisfy that need by the way of a function.

The basic idea is simple and elegant. It reminded me of the kind of thinking we do when we are trying to find the [number of] subsets of a given set. Imagine that the coins are arranged in piles in some order (any order will do) according to their denominations: a pile of 50¢ coins, a pile of pennies, a pile of dimes, and so on.

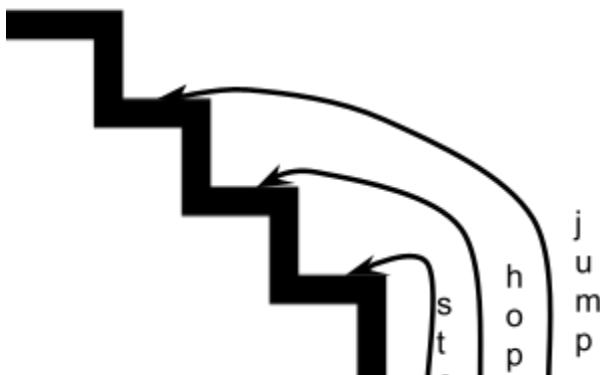


We are handed an amount 'a' and asked to find the number of ways to change that amount using these coin piles that are arranged in some order. Start with maintaining a `count` of the number of ways in which we can change the amount. We will have to take one of the two choices:

- 1) Either take a coin from the first pile of denomination, say  $d_1$ , and see if the reduced amount,  $a - d_1$  can be changed using all the piles, or
- 2) Reject the denomination  $d_1$  and do 1) with the next pile in order.

If and when the amount remaining to be changed runs down to zero, we have found a way to change and we increment the `count`.

Note that we are looking for sets of coins, not their sequences. Thus, in order to change 6¢ it does not matter if you first give a penny and then a nickel or a nickel first and then a penny – both those ways are the same.



Perhaps it is not apparent that this inductive way solves the problem completely in that we

do not overcount or undercount the number of ways.

An analogy may help. Consider a staircase shown here. You can either take a step (one stair at once), a hop (two stairs at once), or a jump (three stairs at once). In how many ways can you climb the staircase with 'n' stairs (once you hope you can't step and once you jump you can only jump)?

It seems more natural to think that if we decide to **not** take a step, we only have hops and jumps to scale the entire staircase.

Perhaps it is clear that once we reject a coin of a particular denomination from our piles of coins arranged according to their denomination, we never choose it again.

A little reflection will let us implement the algorithm thus:

```
function change(amt) {
  function value(denom_id) {
    return denom_id === 1
      ? 50
      : denom_id === 2
      ? 25
      : denom_id === 3
      ? 10
      : denom_id === 4
      ? 5
      : 1;
  }

  function change_using(a, denom_id) {
    return a === 0
      ? 1
      : a < 0
      ? 0
      : denom_id === 6
      ? 0
      : change_using(a - value(denom_id), denom_id) +
        change_using(a, denom_id + 1);
  }

  return change_using(amt, 1);
}
```

The `change_using` function describes a tree recursive process. With `amount = $1`, and values:



denom_id	1	2	3	4	5
value	50	25	10	5	1

here is the function call tree:

```
change(100)
- change_using(100, 1)
  - change_using(50, 1)
    - change_using(0, 1)
    - change_using(50, 2)
      - change_using(0, 2)
      - change_using(50, 3)
        - change_using(0, 3)
        - change_using(50, 4)
          - change_using(0, 4)
          - change_using(50, 5)
            - change_using(0, 5)
            - change_using(50, 6)
  - change_using(100, 2)
...
```

Can we convert this into an iterative process? Can we find some trick in this case (like we found one for [fib\\_iter above](#))? The book leaves it as a challenge to readers.

This is quite a challenge. For starters, I tried to find a pattern. Here is a table of the amount to change (using 50¢, 25¢, 10¢, 5¢, and 1¢ coins) and number of ways to do so:

Amt	0-4	5-9	10-14	15-19	20-24	25-29	30-34	35-39	40-44	45-49	50-54	55-59	60-64	65-69
#	1	2	4	6	9	13	18	24	31	39	50	62	77	93

The number of ways exhibit a “jump” at every multiple of 5. This seems to be because of the peculiar coin denominations that we have, for if we had coins of denominations 45¢, 17¢, 7¢, 2¢, and 1¢, we will get the following table:

Amt	0-1	2-3	4-5	6	7	8	9	10	11	12	13	14	15	16
#	1	2	3	4	5	6	7	8	9	10	11	13	14	16

This appears pretty “random”.

The function “number of ways to change(n)” seems to grow more slowly compared to the `fib(n)` function.

**[TODO] Thus, as of now, I am not able to solve this problem!**

21 February 2023

## 1.2 Functions and Processes They Generate

- Functions generate computational processes. This is a nuanced way of looking at what running a function on a real hardware machine entails.

### 1.2.1

- Two implementations of the canonical factorial implementations.
  - Observe the code and the “shape” of the processes that these two implementations generate.
- **Linear Recursion and Iteration:** This is just beautifully described. I love how they differentiate between a recursive *process* and iterative *process*. A *recursive process has a chain of deferred operations* (in that after a function call is made, there is an “aftermath” of that call). An *iterative process, on the other hand, does not have any deferred operations*.
- The idea of an iterative “process” can be realized *without* the construct of a loop that is found in a traditional imperative language. This gives a different perspective about the processes that functions entail.
- The example considered in the text is the canonical example of recursion as it is taught elsewhere: the factorial function.
- Example of an *iterative process* realized by function calls: Although factorial function is a canonical example, I thought of my own example. Consider this implementation of finding the sum of the first n natural numbers (without using the great Gauss closed-form formula):

```
function sum(n) {  
  function sum_iter(s, counter) {  
    return counter > n  
      ? s  
      : sum_iter(s + counter, counter + 1);  
  }  
  return sum_iter(0, 0);  
}  
sum(5); // => 15
```

- Another example of an **iterative process** realized by function calls: Finding the  $n^{th}$  Fibonacci number.

```
function fib(n) {
  function fib_iter(f1, f2, counter) {
    return counter === n
      ? f1
      : fib_iter(f2, f1 + f2, counter + 1);
  }
  return fib_iter(1, 1, 1);
}
fib(7); // => 13
```

- Example of a **recursive process** realized by function calls: (again, I am avoiding the canonical factorial function. Here is a recursive “process” for the sum function above:

```
function sumr(n) {
  return n === 0
    ? 0
    : n + sumr(n-1); // there are “deferred” operations!
}
sumrec(10); // => 55
```

- Carrying out this recursive process requires that the interpreter keep track of the operations to be performed later on. In the computation of  $n!$  the length of the chain of deferred multiplications, and hence the amount of information needed to keep track of it, grows linearly with  $n$  (is proportional to  $n$ ), just like the number of steps. Such a process is called a **linear recursive process**.
- (This entire section is so fabulous and it reminds me of my own essay: [Recursion and Iteration](#).) I am going to reproduce it verbatim:
  - One reason that the distinction between process and function may be confusing is that most implementations of common languages (including C, Java, and Python) are designed in such a way that the interpretation of any recursive function consumes an amount of memory that grows with the number of function calls, even when the process described is, in principle, iterative. As a consequence, these languages can describe iterative processes only by resorting to special-purpose “looping constructs” such as `do`, `repeat`, `until`, `for`, and `while`. **The implementation of JavaScript we shall consider in chapter 5 does not share this defect.** It will execute **an iterative process in constant space, even if the iterative process is described by a recursive function**. An implementation with this property is called tail-recursive.<sup>3</sup> **With a tail-recursive implementation, iteration can be expressed using the ordinary function call mechanism, so that special iteration constructs are useful only as syntactic sugar.**<sup>4</sup> (Perlisism: “Syntactic sugar causes the cancer of the semicolon.”)

- We can easily see the stress that functions bear in this paradigm (more commonly known as the functional programming paradigm). Such programming languages even consider loops unnecessary since function calls can achieve the same rather painlessly.
- An implementation that provides the support for calling “**recursive functions that represent iterative processes**” is called **tail-recursive** implementation.

[Home work] **Exercise 1.9 and 1.10**. Also, reread this section (1.2.1).

### Exercise 1.9

Each of the following two functions defines a method for adding two positive integers in terms of the functions `inc`, which increments its argument by 1, and `dec`, which decrements its argument by 1.

```
// SICP JS 1.2.1
```

```
function inc(x) {
  return x + 1;
}
function dec(x) {
  return x - 1;
}

function plus(a, b) {
  return a === 0 ? b : inc(plus(dec(a), b));
}

function plus(a, b) {
  return a === 0 ? b : plus(dec(a), inc(b));
}

plus(4, 5);
```

Using the substitution model, illustrate the process generated by each function in evaluating `plus(4, 5)`; . **Are these processes iterative or recursive?**

#### **Solution.**

We depict the shape of the function:

```
function plus(a, b) {
  return a === 0 ? b : inc(plus(dec(a), b));
}
```

using the substitution model:

```
plus(4, 5)
inc(plus(dec(4), 5))
```

```

inc(plus(3, 5))
inc(inc(plus(dec(3), 5)))
inc(inc(plus(2, 5)))
inc(inc(inc(plus(dec(2), 5))))
inc(inc(inc(plus(1, 5))))
inc(inc(inc(inc(plus(dec(1), 5)))))
inc(inc(inc(inc(plus(0, 5)))))
inc(inc(inc(inc(5))))
inc(inc(inc(6)))
inc(inc(7))
inc(8)
9

```

Since there are “deferred operations” in that the `inc` function needs to do its work after the last application of `plus` returns. This is a recursive process.

And now we depict the shape of the function:

```

function plus(a, b) {
    return a === 0 ? b : plus(dec(a), inc(b));
}

```

using the substitution model:

```

plus(4, 5)
plus(dec(4), inc(5))
plus(3, 6)
plus(dec(3), inc(6))
plus(2, 7)
plus(dec(2), inc(7))
plus(1, 8)
plus(dec(1), inc(8))
plus(0, 9)
9

```

Clearly, the shape of the second function suggests an iterative process implemented by recursive function. It can be optimized to be a tail-recursive process.

### [Exercise 1.10](#)

The following function computes a mathematical function called Ackermann's function.

```

function A(x, y) {
    return y === 0
        ? 0
        : x === 0
        ? 2 * y
        : y === 1

```

```

        ? 2
        : A(x - 1, A(x, y - 1));
}

```

What are the values of the following statements?

`A(1, 10);`

`A(2, 4);`

`A(3, 3);`

Consider the following functions, where `A` is the function declared above:

```

function f(n) {
    return A(0, n);
}
function g(n) {
    return A(1, n);
}
function h(n) {
    return A(2, n);
}
function k(n) {
    return 5 * n * n;
}

```

Give concise mathematical definitions for the functions computed by the functions `f`, `g`, and `h` for positive integer values of `n`. For example, `k(n)` computes  $5n^2$ .

### Solution.

Quite a function, this Ackerman function. Applying the substitution model will help finding the value in which an invocation of this function results; but it is clearly tedious. Is there a better way? Let me first try with `A(1, 1)`:

`A(1, 1)`

2

`A(1, 2)`

`A(0, A(1, 1))` // from above

`A(0, 2)`

4

`A(1, 3)`

`A(0, A(1, 2))`

`A(0, 4)` // from above

8

`A(1, 4)`

```
A(0, A(1, 3))
A(0, 8) // from above
16
```

```
A(1, 5)
A(0, A(1, 4))
A(0, 16) // from above
32
```

```
A(1, 6)
A(0, A(1, 5))
A(0, 32) // from above
64
```

```
A(1, 7)
A(0, A(1, 6))
A(0, 64) // from above
128
```

```
A(1, 8)
A(0, A(1, 7))
A(0, 128) // from above
256
```

```
A(1, 9)
A(0, A(1, 8))
A(0, 256) // from above
512
```

```
A(1, 10)
A(0, A(1, 9))
A(0, 512) // from above
1024
```

```
// I admit, however, that doing it without substituting previous
// results was much more tedious
```

```
// This suggests that even though computers do repetitive tasks well,
// one should "cache" the results
```

```
// To get A(2, 4), let's start as below:
```

```
A(2, 1)
2
```

```

A(2, 2)
A(1, A(2, 1))
A(1, 2) // from above
4 // from more above

A(2, 3)
A(1, A(2, 2))
A(1, 4) // from above
16 // from above

A(2, 4)
A(1, A(2, 3))
A(1, 16) // from above
65536 // extrapolating from A(1, n) above

A(2, 5)
A(1, A(2, 4))
A(1, 65536) // from above
 $2^{2^{16}} = 2^{65536}$  // extrapolating from A(1, n) above

// Quite a function! I won't even be able to imagine A(2, 6)

// Let's start with A(3, 1) in the hope of making sense of A(3, 3)

A(3, 1)
2

A(3, 2)
A(2, A(3, 1))
A(2, 2) // from above
4 // from above

A(3, 3)
A(2, A(3, 2))
A(2, 4) // from above
65536 // from above

A(3, 4)
A(2, A(3, 3))
A(2, 65536) // from above
//  $2^{A(2, 65535)}$  // huge!

```

The [bi]function is several functions in one!



$$f(n) = A(0, n)$$

This function clearly doubles the argument passed. Thus,  $f(n) = 2n$ .

$$g(n) = A(1, n)$$

$$\therefore g(n) = A(0, A(1, n-1)) = A(0, g(n-1)).$$

Since  $A(0, n) = f(n) = 2n$ , we get

$$g(n) = A(0, g(n-1)) = 2g(n-1)$$

We also know that  $g(1) = 2$ .

This is a recurrence relation. We can solve it by repeatedly substituting and we'll get:

$$\begin{aligned} g(n) &= 2g(n-1) = 2^2g(n-2) = \dots = 2^{n-1}g(n-(n-1)) = 2^{n-1}g(1) \\ &= 2^n. \end{aligned}$$

$$h(n) = A(2, n) = A(1, A(2, n-1)) = A(1, h(n-1)) = g(h(n-1))$$

and we have  $h(1) = 2$

Since  $g(n) = 2^n$ , we have  $h(n) = 2^{h(n-1)} = 2^{2^{h(n-2)}} = 2^{2^{2^{h(n-3)}}}$  and finally there will be  $n$  2's in the exponent.

[The Ackermann](#) function is really complex. In Knuth's [up-arrow notation](#),

$$A(2, n) = h(n) = 2 \uparrow \uparrow n$$

Mathematician R. L. Goodstein has called repeated exponentiation [tetration](#).

09 February 2023

Exercises 1.6, 1.7, and 1.8 are important (they also take time). Let's finish those by 12 February 2023.

#### 1.1.8 Functions as Black-box Abstractions

- It's crucial that each function accomplishes an identifiable task that can be used as a module while defining other tasks. By assuming the existence of the other functions that a function we are defining uses we achieve **functional abstraction**.
- This idea is also known as API. Purely functional abstraction is less stringent than an API, but the concept of "hiding the implementation details" is common to both. It does not matter how the square of an argument is calculated as long as we know that it is.

#### Local Names

- The formal parameters of a function are local to its definition. They do not muddy the "global namespace".
- Clearly, lambda calculus is intended here because lambda calculus formalizes function application to actual arguments.
- Bound names and Free names (paraphrasing): A [formal] parameter of a function has a very special role in the function declaration, in that **it doesn't matter what name the parameter has. Such a name is called bound**, and we say that the **function declaration binds its parameters**. The meaning of a function declaration is unchanged if a bound name is consistently renamed throughout the declaration. **If a name is not bound, we say that it is free.**
  - Scope: **The set of statements for which a binding declares a name** is called the scope of that name.
  - The meaning of a function is independent of its choice of bound names (one can carefully replace a bound name by something else and it will not change the meaning of the function), but not independent of its free names.

[Home Work] Complete 1.1.8 (reading) and discuss next time.

06 February 2023

#### 1.1.7 Square-roots by Newton's method

- Difference between declarative knowledge – what is (that's how mathematical functions are expressed) and imperative knowledge – how (that's how functions are *implemented* in computers).
- There has been some progress on "automatically" generating computer functions from their "specification" or declaration, but, in general, this is not easy.
- Newton's method of successive approximations. I have implemented it [here](#) (P1.1).
- I really like how they have created this computer function that feels declarative:

```
function sqrt_iter(guess, x) {  
  return is_good_enough(guess, x)
```

```

        ? guess
        : sqrt_iter(improve(guess, x), x);
}

```

It simply “declares” at a high-level what we want to do, not so much of how we want to do it. The “details” are hidden away or, more precisely, abstracted out of this strategy. This is really good code:

```

// SICP JS 1.1.7

function abs(x) {
    return x >= 0 ? x : - x;
}

function square(x) {
    return x * x;
}

function is_good_enough(guess, x) {
    return abs(square(guess) - x) < 0.001;
}

function average(x, y) {
    return (x + y) / 2;
}

function improve(guess, x) {
    return average(guess, x / guess);
}

function sqrt_iter(guess, x) {
    return is_good_enough(guess, x)
        ? guess
        : sqrt_iter(improve(guess, x), x);
}

sqrt_iter(1, 0.5);

```

### **Exercise 1.6**

Alyssa P. Hacker doesn't like the syntax of conditional expressions, involving the characters `?` and `:`. "Why can't I just declare an ordinary conditional function whose application works just like conditional expressions?" she [asks](#). Alyssa's friend Eva Lu Ator claims this can indeed be done, and she declares a conditional function as follows:

```

function conditional(predicate, then_clause, else_clause) {

```

```
    return predicate ? then_clause : else_clause;
}
```

Eva demonstrates the program for Alyssa:

```
conditional(2 === 3, 0, 5);
=> 5
conditional(1 === 1, 0, 5);
=> 0
```

**Delighted, Alyssa uses conditional to rewrite the square-root program:**

```
function sqrt_iter(guess, x) {
    return conditional(is_good_enough(guess, x),
                      guess,
                      sqrt_iter(improve(guess, x),
                                  x));
}
```

What happens when Alyssa attempts to use this to compute square roots? Explain.

### **Solution.**

This is such a good question! The real exercise is to predict the behavior of this program (which means one should not run the program and then explain its behavior).

Introduction of a function call combined with the applicative-order evaluation of the interpreter would result in an infinite recursion. The program will not terminate for any input.

A function call interpretation is key here. Consider an applicative-order interpreter. Such an interpreter evaluates all the expressions provided for actual arguments and then applies the called function (`sqrt_iter`) to those arguments. Before calling `conditional`, the interpreter must

- 1) evaluate the actual value of `predicate` which is the value returned by the `is_good_enough` function with arguments `guess` and `x` (this will be either `true` or `false`).
- 2) assign the current value of `guess` to the parameter `then_clause`.
- 3) evaluate the actual value of `else_clause` which is the value returned by `sqrt_iter` again with an *improved* value of `guess` – this results in another call to the same function without any base case  $\Rightarrow$  an endless recursion.

With an applicative-order interpreter endless recursion should happen for every input. With a normal-order interpreter, however, if the `predicate` is `true`, then the `else_clause` need not be evaluated, causing the recursion to end at some point.

**Program execution result.** It seems my answer is right. The interpreter says: **Line 34: The function `sqrt_iter` has encountered an infinite loop. It has the infinite cycle: (guess: 5), (x: 25) -> (guess: 5), (x: 25).**

### Exercise 1.7

The `is_good_enough` test used in computing square roots will not be very effective for finding the square roots of very small numbers. Also, in real computers, arithmetic operations are almost always performed with limited precision. This makes our test inadequate for very large numbers.

Explain these statements, with examples showing how the test fails for small and large numbers. An alternative strategy for implementing `is_good_enough` is to watch how `guess` changes from one iteration to the next and to stop when the change is a very small fraction of `guess`. Design a square-root function that uses this kind of end test. Does this work better for small and large numbers?

#### **Solution.**

Consider our test which is in the function `is_good_enough`:

```
function is_good_enough(guess, x) {  
    return abs(square(guess) - x) < 0.001;  
}
```

This means that if the absolute difference between the target number (`x`) and the square of our `guess` is less than 0.001, we consider the `guess` to be good enough.

Now consider a small number `x = 35e-16` (i.e. 0.00000000000000035) whose square-root we wish to know. In our iterative process, if our `guess` is such that its square is close to our tolerance or margin of 0.001, then `square(guess) - x` will also remain close enough to 0.001 because `square(guess) >> x`. In other words, however small `x` is, the iteration will end around the square-root of 0.001 returning an erroneous result ( $\approx 0.0316$ ). The correct square-root of `x = 35e-16` is `5.916079783099616e-8`.

What if we made the margin or tolerance smaller? Say `0.0000001` or `1e-7` instead of 0.001 or `1e-3`? That will make the problem less severe and we may be able to find square-roots of more (smaller) numbers. But the issue will resurface soon. Therefore, a redefinition of our test like below only goes a little farther:

```
function is_good_enough(guess, x) {  
    return abs(square(guess) - x) < 1e-7;  
}
```

This will run into the same issue in that we'll erroneously stop the iteration around the values of square-roots that are about  $3.16e-3$ .

What is the basic issue with our test function? ***It defines a tolerance in absolute terms.***

Because of an infinitude of numbers, there will always be a number that beats an absolute margin or tolerance no matter how small. The problem is exacerbated because of an inherent limitation of the finite machine that is our silicon computer that is tasked with representing infinity of numbers! As we know, the infinity of real numbers is such that there is no successor of a given real number. Or, in other words, one can always find a number that lies between any two numbers however close. ***For a machine that only has a few “bits” to represent infinitely many numbers, there is no choice but to define a successor of a real number, losing precision in the process!*** This is what happens for numbers like  $35e-16$  that are smaller than 1.

For numbers much larger than 1, a different kind of problem surfaces. To demonstrate that problem, I tried the following numbers in succession:  $1e10$ ,  $1e20$ ,  $1e30$ ,  $1e40$ ,  $1e50$ . For the last number, the interpreter shows the following message: **The function `sqr_iter` has encountered an infinite loop. It has an infinite cycle: (guess:  $1e+25$ ), (x:  $1e+50$ ) -> (guess:  $1e+25$ ), (x:  $1e+50$ ).**

This demonstrates the inability of our code to fail to progress toward its goal. The algorithm starts with a guess and hopes to make a gradual, but definite progression toward the goal. This is called “optimization.” And since the implementation is recursive, the physical implementation of it on a stack-based Von-Neumann computer is, again, limited by the capacity of a finite-resource machine. We don't have the resources to apply a function an infinite number of times!

The reason we run out of the stack space (i.e. the resource required for any function application) is that we fail to detect that we are making progress toward a goal. That is, at some point, a function application is exactly identical to the previous function application: their arguments are the same and they return a value that is more than the preset margin:  $0.001$ . When we get into such a *hole*, we are stuck. The reason is related (again) to the finiteness of the machine on which we run the program: the average of two numbers,  $x$  and  $y$ , fails to get in between them, it gets “rounded” to either extreme.

A closer examination of why we get stuck suggests a way out – we see a silver lining to a dark cloud. What if we abandon an absolute comparison in favor of a relative comparison across successive calls? Since our hope and effort are that we make “progress” on every function call, we should abandon the effort when the relative progress is too small to notice:

```
function is_close_enough(x, y) {  
    return abs(x - y) < 0.001 * x;  
}
```

And apply the function thus:

```
function sqrt_iter(guess, x) {  
  return is_close_enough(guess, improve(guess, x))  
    ? guess  
    : sqrt_iter(improve(guess, x), x);  
}
```

This tells us if we are making any noticeable improvement by measuring a relative change rather than an absolute one. This makes our code much more robust!

```
square(sqrt(42e100));  
=> 4.2016359946024356e+101
```

The original code, on the other hand, fails with the infinite loop message:

```
function sqrt_iter(guess, x) {  
  return is_good_enough(guess, x)  
    ? guess  
    : sqrt_iter(improve(guess, x), x);  
}
```

A subtle change indeed!

Upon a closer examination, I realized something about the improved [version](#) of `sqrt_iter`. In all the recursive calls except when the current `guess` is close enough to the square-root, we will end up making two *identical* calls to `improve(guess, x)`. **Both these calls apply the same function to the same actual arguments and return the same value!** Why should we incur that cost? First off, is there any value in thinking about it this way? Or is it thinking too prematurely about “efficiency”? Either way, I think at least mentioning this point is worth the effort. Therefore, consider this reincarnation of `sqrt_iter`:

```
function sqrt_iter(guess, x) {  
  const improved_guess = improve(guess, x);  
  return is_close_enough(guess, improved_guess)  
    ? guess  
    : sqrt_iter(improved_guess, x);  
}
```

### [Exercise 1.8](#)

Newton's method for cube-roots is based on the fact that if  $y$  is an approximation to the cube-root of  $x$ , then a better approximation is given by the value

$$\frac{\frac{x}{y^2} + 2y}{3}$$

Use this formula to implement a cube-root function analogous to the square-root function. (In section [1.3.4](#) we will see how to implement Newton's method in general as an abstraction of these square-root and cube-root functions.)

### Solution.

Newton's method uses calculus to come with the formula for better approximation. That is a separate exploration that I will write about a little later.

For the exercise alone, we can find a good way to implement the approximation. But before that, let me ask a question to myself: As this exercise asserts, in the case of the cube-root of  $x$ , if  $y$  is

a guess, then a better guess is  $\frac{\frac{x}{y^2} + 2y}{3}$ . What is the equation for a better guess in the case of square-root of  $x$ ?

This is “reverse-mathematizing”. We have already coded the method to find the square-root. We can observe that a mathematical expression for a better guess (in the code we wrote) for

$y = \sqrt{x}$ . It turns out that that expression is  $\frac{\frac{x}{y} + y}{2}$ .

Several questions arise, one of which is: Can the same thinking apply to finding a better approximation of cube-root of  $x$ ? In a preceding section we said that to get a better approximation of the square-root of  $x$ , we simply take the **average** of  $y$  (current approximation) and  $\frac{x}{y}$ . The *sicpjs* text does not explain why one should take the average of the two (other than

it being “intuitive”). The reason that (when  $y$  is an approximation)  $\frac{\frac{x}{y} + y}{2}$  is a better approximation lies in [Newton's method of finding roots](#). Newton's method is the reason that in the case of cube-root  $\frac{\frac{x}{y^2} + 2y}{3}$  is a better approximation.

Perhaps we could simply improve the `improve` function to accommodate a better approximation while finding the cube-root of  $x$ , the rest of the code remains the same. One rather straightforward way to code that is:

```
function improve3(guess, x) {  
    return (x / square(guess) + 2 * guess) / 3;  
}
```

Of course, this function is not “abstract” enough.



01 February 2023

### 1.1.6 Conditional Expressions and Predicates:

- Case analysis.
- `<predicate> ? <consequent-expression> : <alternate-expression>`
- Predicate here is used in the sense of a mathematical proposition: either `true` or `false`.
- These can be chained.
- Primitive predicates for numbers: `>`, `<`, `===`, `>=`, `<=`, `!=="`
- Logical predicate operators for combining predicates themselves:
  - Conjunction: `<predicate-1> && <predicate-2>`
  - Disjunction: `<predicate-1> || <predicate-2>`
  - Negation: `! <predicate>`

### Exercises

#### [Exercise 1.3](#)

Declare a function that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

```
function sum_of_squares_2(a, b, c) {  
    return a > b  
        ?  
        (b > c ? sum_of_squares(a, b) : sum_of_squares(a, c))  
        :  
        (a > c ? sum_of_squares(a, b) : sum_of_squares(b, c));  
}
```

The prescribed solution incurs also two comparisons, but uses a somewhat complicated identity to reach the correct solution. I like my solution better!

#### [Exercise 1.5](#)

Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using applicative-order evaluation or normal-order evaluation. He declares the following two functions:

```
function p() {  
    return p();  
}  
  
function test(x, y) {  
    return x === 0 ? 0 : y; //  
}
```

Then he evaluates the statement:

```
test(0, p());
```

---

I discussed this with Apoorv and he made a comment while studying the code of `test(x, y)`: If all Ben wanted to test was whether the interpreter evaluation is applicative-order or normal-order, the following definition should suffice:

```
function test(x, y) {  
    return true || y;  
}
```

while the function application (function call) remains: `test(0, p(0))`;

- if it results in infinite loop, the interpreter is applicative-order,
  - otherwise it is normal-order.
- 

**Question:** What behavior will Ben observe with an interpreter that uses applicative-order evaluation?

**Answer:** Applicative-order evaluates actual arguments eagerly, rather than when they are actually needed. Thus, even though the conditional in the `test` function evaluates to `true` and there is no need to evaluate the actual argument that substitutes `y`, an interpreter employing the applicative-order will end up doing just that. Since calling `p()` results in an infinite loop, the program will not terminate.

**Question:** What behavior will he observe with an interpreter that uses normal-order evaluation?

**Answer:** This would actually continue normally and return `0`. This is because the predicate of the conditional evaluates to `true` when the formal parameter `x` is substituted with the actual argument zero.

Explain your answer. (Assume that the evaluation rule for conditional expressions is the same whether the interpreter is using normal or applicative order: The predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.)

## 01 February 2023

### 1.1.5 The Substitution Model of Function Application

- It is all about replacing the formal parameters by actual arguments at the time of function application (or at the time of the function call).
- Think of it as a token replacement. Wherever the interpreter sees the parameter `a`, it replaces it with the actual argument, e.g. `2`.
- Normal-order evaluation and Applicative-order evaluation
- I tend to call **Normal-order** evaluation *Lazy evaluation* and **Applicative-order** evaluation *Eager evaluation*.

Consider the three function declarations (or definitions) followed by a function call (or function application):

```
function square(x) {
```

```

    return x * x;
}

function sum_of_squares(x, y) {
    return square(x) + square(y);
}

function f(a) {
    return sum_of_squares(a + 1, a * 2);
}

f(5);

```

Here, when the function call is interpreted by an interpreter that uses substitution model, it substitutes values in a straightforward manner (leaving aside the issues of duplicate names):

```

return sum_of_squares(5 + 1, 5 * 2)
->
((square(5 + 1) + square(5 * 2)))
->
((5 + 1) * (5 + 1) + (5 * 2) * (5 * 2))
->
(6 * 6 + 10 * 10)
->
136

```

An important statement in this section: Over the course of this book, we will present a sequence of ***increasingly elaborate models of how interpreters work***, culminating with in a complete implementation of an interpreter and compiler in chapter [5](#).

- Is it possible to have normal-order evaluation result in a value different from the applicative-order evaluation? There is an exercise for that.

31 January 2023

#### 1.1.4 Compound Functions

Function declaration statement:

```
function name(parameters) { return expression; }
```

Function application expression: `function-expression(arguments)`

(Note that it is a function-expression, not necessarily a function-name.)

## 26 January 2023

### 1.1.3 Evaluating Operator Combinations

[Homework] While discussing this section, we thought of two assignments that Apoorv may enjoy:

- 1) Study the familiar infix notation. You already know it, but we need you to *codify* your understanding. One way to codify a grammar (what distinguishes a valid expression from an invalid one) is **railroad diagrams**. Referring to a good railroad diagram reference, define a railroad diagram for infix notation.
- 2) Write a program to convert infix notation to postfix notation. After you have done it, run it on several inputs and then evaluate the postfix expression (write a program to do that as well) to make sure that your conversion is correct. Can you make only little tweaks to your program of conversion to convert a given infix expression to prefix notation? Compare your “translation algorithm” with the so-called Shunting Yard algorithm.

## 23 January 2023

### 1.1.1 Expressions

- Expression statements – they have “values”
- REPL

### 1.1.2 Naming and Environment

- Names allow us to abstract computational objects away
- Environment helps “memorize” name-object pairs

### 1.1.2 Evaluating the Operator Combinations

- Inherently recursive way of combining and evaluating expressions associated via operators

## 19 January 2023

### 1.1 Elements of Programming

- This echoes John Locke’s paraphrased statements. Every programming language provides
  - primitive constructs,
  - means to combine primitives, and
  - means of abstraction

## 17 January 2023

### Chapter 1 Building Abstractions with Functions

- Computational process, data, programs.
- What we should teach students:

- Not algorithms/data structures; not even mathematical underpinnings
- But two key aspects of software's nature:
  - Understanding that programming languages are much more than mere tools to solve a given problem.
  - Controlling complexity of software projects.
- JavaScript
  - Webpage anatomy: markup (structure), CSS (appearance/style, look), JavaScript (behavior, feel)
  - Inherited from programming languages: Scheme and Self
    - Lexical scoping
    - Dynamic typing
    - First-class functions
  - Interpretation vs. Compilation
    - Please see this [3-minute documentary](#) from the gobbledegook planet!
- John Locke:
  - Combining (simple ideas to form complex ideas)
  - Relations (among ideas simple and complex)
  - Abstraction (remove the idea from its practical/real existence)

## 11 January 2023

We read the preface. Kedar felt the need to introduce Apoorv to the basic structure of a computer. We are going to take a brief look at *Programming from Ground Up* by Jonathan Bartlett. Kedar also thought that reading the introductory articles about  $\lambda$ -calculus may stand in good stead. Find time to read:

- 1) Stanford Encyclopedia of Philosophy [on  \$\lambda\$ -calculus](#).
- 2) Raul Rojas's tutorial [introduction](#) to  $\lambda$ -calculus.

## 7 January 2023

Guy Steele (an influential programmer and programming language designer), in his own foreword, asks a question about Alan Perlis's foreword to SICP:

Is it better to have 100 functions operate on a sort of “Universal data structure” than to have 10 functions operate on 10 different, specialized data structures?

He says we tend to prefer 10 different functions operating on 10 data structures for “human reasons”; we tend to organize ourselves better by giving names to things. Otherwise, he says, it is not too difficult (only somewhat tricky) to write a sort of *universal function*, say  $uf$ , that accepts a “description of any other function  $f$ ” and a variable list of arguments and does the job of that function  $f$ . We are going to see an example of this later.

JavaScript arrow function. Read about it and discuss it [here](#).

### Apoorv's Understanding of Arrow Functions

Arrow functions are unnamed, concise ways to write functions. They are useful when a function needs to be passed in to a method because they can be written inline. They are written as a set of parameters, followed by `=>`, and followed by a function expression or set of statements.

For example, here is an arrow function:

```
x => 5 * (x / 4)
```

This function takes in one parameter, `x`, and returns  $5x/4$ . The initial way to write this arrow function would be:

```
function someName(x) {  
  return 5 * (x / 4);  
}
```

Arrow functions with multiple or no parameters must surround the parameters with parentheses. Additionally, arrow functions may have multiple statements as opposed to an expression. However, the statements must be enclosed in braces, be multiline, and contain a return statement - just like a standard function.

```
(x, y) => {  
  if (x < 1) {  
    x += 1;  
  }  
  return x + y;  
}
```

You might see “double arrow functions” written like this:

```
x => y => x * y
```

This is just an example of [currying](#). Instead of taking two parameters `x` and `y`, this arrow function is taking a parameter `x`, calling a function which takes a parameter `y`, and from inside this second function, returning  $x \cdot y$ . The standard function for this would be:

```
function curryOuter(x) {  
  return function curryInner(y) {  
    return x * y;  
  }  
}
```

The advantage of currying is not apparent with this function. Currying allows you to compose functions and ensure that operations are carried out in a specific order. Arrow functions just allow you to do that more concisely.

Investigate this [statement](#): And it is only slightly tricky to design a data structure capable of describing any computation whatsoever.

Kedar to read and comment on Steele's paper: Growing a language.

Let's list action items!

A-I

- 1) **Apoorv**, take a look at: <https://www.macs.hw.ac.uk/~greg/publications/gjm.idc15.pdf>

5 January 2023

Please read the foreword 2: <https://sourceacademy.org/sicpjs/foreword02> and discuss.

## Required Readings

A Classic Paper – Lisp: A Language for Stratified Design (Kedar Annotates)

P1.0 Introduction

The paper is available on the Web at: <https://dspace.mit.edu/handle/1721.1/6064>. Below is a (short) annotated version of the paper.

We annotate the paper by first copying verbatim a part of the paragraph of interest and then annotating it.

We exhibit programs that illustrate the power of Lisp as a language for expressing the design and organization of computational systems. The examples are chosen to highlight the importance of *abstraction* in program design and to draw attention to the use of procedures to express abstractions.

“Stratified” means arranged in strata – *layers* or *levels*. A CS term for layer is *abstraction*. Lisp promotes abstractions. It is claimed that choosing abstractions carefully improves the quality of software. A programming language should promote building abstractions.

Any programming language provides at least three things:

- primitives
- means to combine those primitives to create sort of “higher level primitives”
- means to name such combinations so that they can be used or manipulated as primitives themselves

This is akin to mathematics or (in a somewhat less stringent manner) even spoken languages. In mathematics, we have axioms which are given, operations to deductively combine those axioms, and theorems and corollaries which are just another form of what we are given. Proven theorems can then be used to prove other theorems. At this point, proven theorems become indistinguishable from given axioms.

Abstractions of components separate their specification from their implementation. This ought to provide greater flexibility in design. An implementation of a clearly specified component can be replaced by another implementation that is qualitatively or quantitatively better.

The paper says that some programming languages (it was 1987, so the mainstream languages were Fortran and C) limit our ability to make abstractions by imposing some restrictions on procedural abstractions. These restrictions seem artificial or arbitrary:

- Having to provide the name of a procedure at the “callsite” in lieu of its definition. (Exactly why is naming a procedure so that it can be “called” a restriction?)
- Not letting procedures to be returned as values from other procedures. This makes “data” different from “procedures” – this is unnecessary. Nowadays (in 2022) when such a limitation is *not* there in a programming language, people say that in that language “procedures or functions are first-class citizens”.
- Not letting procedures be elements of collections.

The well-publicized programming methodology of top-down, structured design produces systems that are organized as trees. Following this methodology, a system is designed as a predetermined combination of parts that have been carefully specified to be combined as determined. Each of the parts is itself designed separately by this same process. The methodology is flawed:

...

In layered or stratified design, the layer 0 is made of primitives. Each subsequent layer is designed as a standalone abstraction; it creates additional primitives. Those primitives are available to *all* the subsequent layers. At a layer, it is impossible and unnecessary to know which layer above it a primitive comes from.



Each level of a stratified design may be thought of as a **specialized language** with a variety of primitives and means of [their] combination appropriate to that level of detail. ...

On the other hand, the structured approach looks at a system as a tree (or is it a graph?) of components. The components have dependencies and they are usually like carefully designed, well-specified black boxes. Every blackbox is expected to satisfy its contract, its “interface”. Good, clean implementations of such interfaces provide flexibility in that implementations can be replaced without breaking the system. These are all *good things*.

Thus, the structured approach does have some favorable things. Then why does the paper claim that it is a *flawed* methodology? And why is layered design better than structured design? Here is one reason:

If a system is to be robust, it must have **generality more than is needed for the particular application**.

Structured design promotes ad-hoc (made for a specific purpose) code. Layered design promotes generalizable, composable components that are loosely coupled. By combining them conveniently, a new application can be conceived relatively easily and changes can be addressed relatively quickly.

**The real power of Lisp is that its unrestricted abstractions support the construction of new languages, greatly facilitating the strategy of stratified design.**

The above statement (creating newer languages from Lisp) needs some explanation that we'll come back to. Lisp provides abstraction support deeply. The paper discusses it at various levels. We start with a support to abstract out procedures or functions.

## P1.1 Abstractions as Procedures

Consider a rudimentary square-root finding procedure in JavaScript:

```
//find the square root of a given number
function sqRoot(x) {
  let epsilon = 1e-6;
  let low = 1.0;
  let high = x;
  if (x < 1.0) {
    low = 0.0;
    high = 1.0;
  }
```

```

}
let guess = (low + high) / 2;
let d = guess * guess - x; // line #10
do {
  // console.log("guess:", guess, "delta: ", d);
  if (Math.abs(d) < epsilon) {
    break;
  }
  if (d < 0) { // guess is too small
    low = guess;
  } else {
    high = guess;
  }
  guess = (low + high) / 2;
  d = guess * guess - x; // line #22
} while (Math.abs(d) > epsilon);
return guess;
}

```

Running this program for some input produces the following output:

```

let x = 64;
console.log("non-parameterized: square-root(", x, ") = ", sqRoot(x));
//non-parameterized: square-root( 64 ) = 8.000000052154064

```

Although this is a straightforward iterative procedure that converges for most of the positive inputs, it has limitations. There are no reusable, composable parts in it. For a cube-root finding procedure, we will have to repeat most of the code.

How can the procedure be made *reusable*? By thinking about this procedure a little differently. What is the procedure really doing? Looking at lines 10 and 22 may provide some cue. It is making some computation *converge* iteratively. In the square-root finding process the computation we are trying to converge is the *squaring*. But squaring is not the only function that can utilize this “converging service”. How about a function finding cube-root? Wouldn’t that procedure need an identical converging service for a cubing function? Of course, yes. And functional languages encourage such thinking by allowing functions (computations) to be passed to other functions. Here is a “refactoring” of code that implements that idea:

```

//help “converge” a function f that accepts an initial value x
function converge(f, x) {
  let epsilon = 1e-6;
  let low = 1.0;
  let high = x;
  if (x < 1.0) {
    low = 0.0;
    high = 1.0;
  }
  let guess = (low + high) / 2;
  let d = f(guess) - x;

```

```

do {
  // console.log("guess:", guess, "delta: ", d);
  if (Math.abs(d) < epsilon) { // we found the fixed point of f
    break;
  }
  if (d < 0) { // guess is too small
    low = guess;
  } else {
    high = guess;
  }
  guess = (low + high) / 2;
  d = f(guess) - x;
} while (Math.abs(d) > epsilon);
return guess;
}

//passes the squaring function literal to the converge function to find
square-root
function paramSqRoot(x) {
  return converge(function (p) {
    return p * p;
  }, x);
}

//passes the cubing function literal to the converge function to find
cube-root
function paramCubeRoot(x) {
  return converge(function (p) {
    return p * p * p;
  }, x);
}

```

This produces the following output:

```

let x = 64;
console.log("parameterized: square-root(", x, ") = ", paramSqRoot(x));
console.log("parameterized: cube-root(", x, ") = ", paramCubeRoot(x));
parameterized: square-root( 64 ) = 8.000000052154064
parameterized: cube-root( 64 ) = 3.9999999972060323

```

Thus, we could utilize the same code by what is called “behavior parameterization”. This is how we abstract procedures out. As the paper says, *we decompose methods into essential, useful pieces*.

Can we extend this to any function that behaves like the square-root function?

## P1.2 Stratified Design

This is about Peter Henderson’s [paper](#) named functional geometry. Authors claim that Henderson used *stratified design* to describe M. C. Escher’s woodcut: Square Limit. That paper teaches me something new. This is the strength of functional declarative programming:

The idea that one could ***write an algebraic description, embed it in a functional program, and execute it directly*** was not new [even back then in 1982, when version 1 of the paper was written]. But it was not considered a practical programming technique. Now we know better and many examples exist of how practical it can be to simply write ***denotations of what is to be constructed***, rather than to write ***algorithmic descriptions of how to perform the construction***. [What, rather than How].

Algebraic denotations suggest a ***language***! In algebra, we make up rules that deal with abstract things and we use these rules consistently to manipulate those things. The language is made of symbols. A ***similar, denotational description can be applied to complex pictures***:

***A picture is an example of a complex object that can be described in terms of its parts. Yet, a picture needs to be rendered on a printer or screen by a device that expects [to be given] a concrete sequence of commands. Programming that sequence of commands directly is much harder than having an application generate the commands automatically from the simpler, denotational description.***

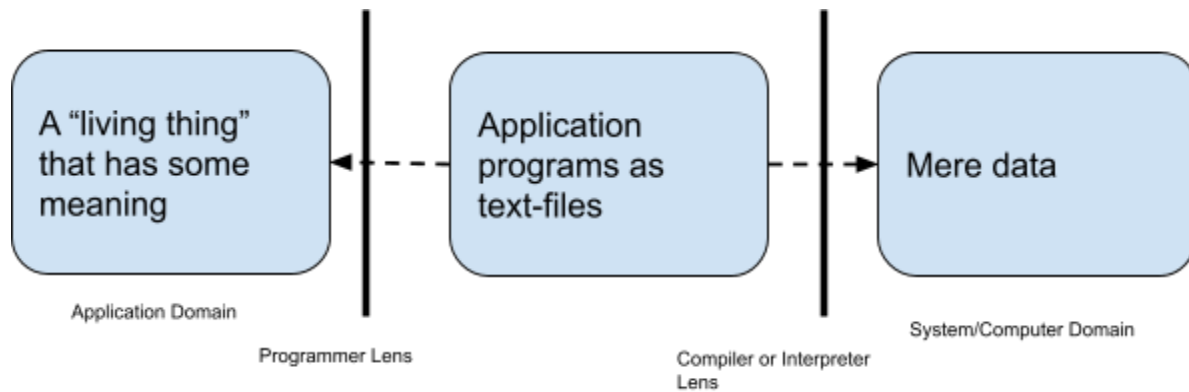
< I am going to come back to the remainder of this section >

### P1.3 Metalinguistic Abstraction

Wikipedia defines it thus: In computer science, metalinguistic abstraction is the **process of solving complex problems by creating a new language or vocabulary to better understand the problem space**.

IMO, the first part of this section is rather confusing. Whereas I understand that a large system may combine object-oriented, functional, and procedural programming elements in a (hopefully) coherent whole, it is unclear why it suddenly starts referring to compilation/interpretation. The transition is not smooth.

The paper then talks about compilers/interpreters.



An application program has a specific meaning and it solves a specific problem in the application domain. For a programmer, their program is not just a string of characters. But for an interpreter or compiler it is just so, mere data.

I believe this will become clearer during the actual SICP course.

The paper, however, does *not* lucidly bring out what *metalinguistic abstraction* really means. It uses the term “application language” as well as “programming languages”. Are these the same? Wikipedia says that metalinguistic abstraction is about creating new languages.

\_\_\_\_\_ start: sort of sidebar \_\_\_\_\_

I am going to write some thoughts in this sidebar about “metaprogramming”. Some of this is inspired by the [metaprogramming article on Wikipedia](#).

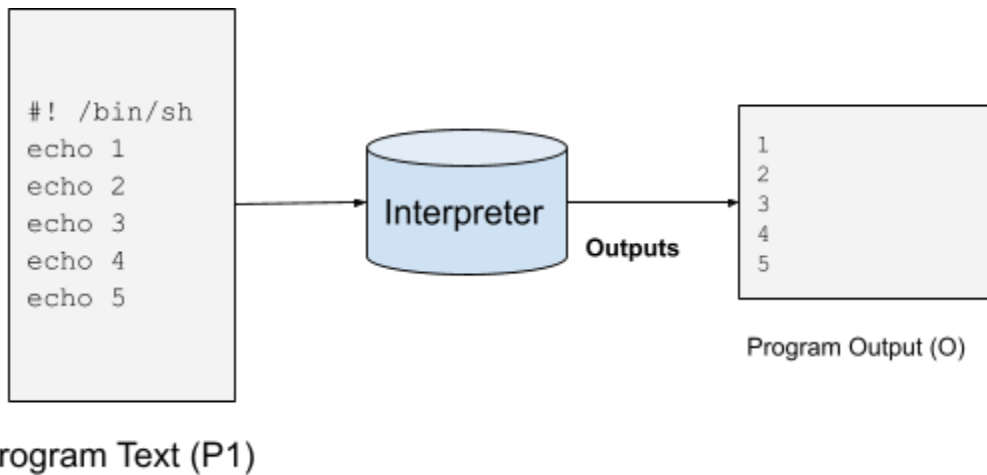
Consider a shell script (there is nothing specific about using a “shell” script – it is applicable to a program written in any general programming language that is Turing-complete – we don’t yet know what “Turing completeness” means, but we’ll press on. For now, a Turing-complete programming language is a general programming language like C or Lisp or Java). It is a *toy-program*:

```
#!/bin/sh
echo 1
echo 2
echo 3
echo 4
echo 5
```

When this program `P1` is interpreted (or run, or executed), the following output is produced:

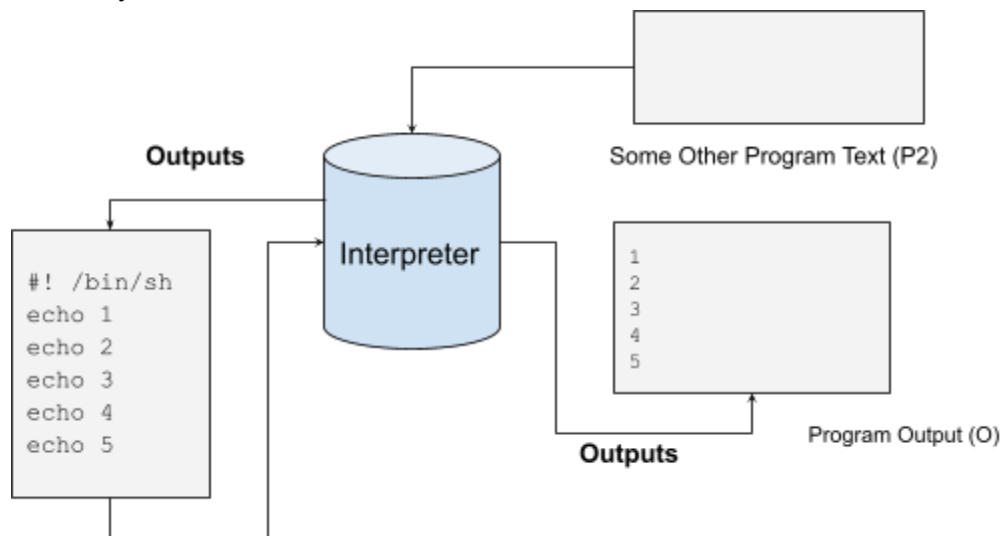
```
1
2
3
4
5
```

Pictorially, it looks like this:



Now, rather than writing it by hand, can one **generate** the program?

Pictorially, it looks like this:



P2 -> interpreter -> P1 -> interpreter -> O

A few moments' reflection shows that we can take a stab at writing the text of the program P2:

We save that program as a text-file named `p2.sh`:

```
#!/bin/sh
echo "#!/bin/sh" > plg.sh # p1 generated
echo "echo 1" >> plg.sh
echo "echo 2" >> plg.sh
echo "echo 3" >> plg.sh
echo "echo 4" >> plg.sh
echo "echo 5" >> plg.sh
```

```
chmod +x plg.sh
```

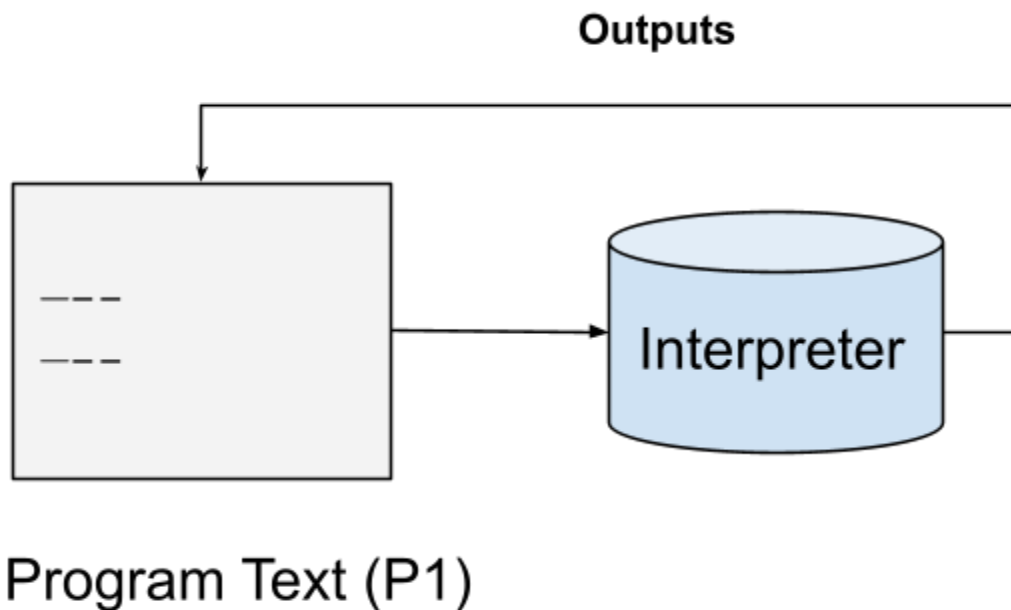
Running `p2.sh` generates a “program” `plg.sh`, which is identical to `p1.sh`. Clearly, when interpreted, both `p1.sh` and `plg.sh` produce the same output. **Generating the same program was our goal.**

We can always write an improved version of P2, `p2i.sh`:

```
#!/bin/sh
echo "#!/bin/sh" > plg.sh # p1 generated
for i in `seq 5`
do
    echo "echo $i"
done
chmod +x plg.sh
```

The **same interpreter** interprets both the programs: P1, P2. Although they have different “meanings” for the programmer in the program domain, for the interpreter, they are just data that it interprets just the same way.

Does this discussion help realizing a special program called **quine** which produces itself as its output when interpreted? Pictorially it looks like:



Not yet!

I am not getting any insight about how to “go about” writing a quine.

Then, it dawned on me that I can “circumvent” the definition of Quine. I wrote a Java program that, when run, prints its source code:

```
package practice;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Quine {
    public static void main(String[] args) throws IOException {
        BufferedReader r = new BufferedReader(new
InputStreamReader(System.in));
        String s;
        while ((s = r.readLine()) != null) {
            System.out.println(s);
        }
        r.close();
    }
}
```

Chat GPT was not impressed and told me:

I apologize for the misunderstanding. It is possible to use this program to create a quine, but it is not a quine itself. To use this program to create a quine, you would need to provide the source code of the program as input when you run it. For example, you could run the program like this:

```
java Quine < Quine.java
```

This would read the source code of the Quine program from the file Quine.java and pass it as input to the Quine program. The Quine program would then read the input from stdin and print it back out, resulting in the source code of the Quine program being printed to the console.

**However, this is not the same thing as the Quine program being a quine itself. A quine is a self-contained program that can output its own source code without requiring any external input.** In the case of this program, it requires external input (the source code of the Quine program) in order to output its own source code.



A Quine is a type of computer program named after the philosopher Willard Van Orman Quine. It is a program that outputs its own source code when run. The concept of a Quine is often used as a thought experiment or **as a test of a programming language's metaprogramming capabilities.**

Quines are interesting because they **demonstrate that it is possible to create a self-contained program that can output its own source code without requiring any external input or information. This is possible because the source code of a program is essentially a representation of the instructions that the program follows when it is run. By including the source code of the program itself as part of the instructions that the program follows, it is possible to create a program that can output its own source code.**

Quines are often used as a way to test the limits of a programming language or to explore the concept of self-reference in computer programs. They are also sometimes used as a form of computer art or as a recreational programming challenge.

-----end: sort of sidebar-----

Lisp Paper Annotated by Apoorv (His Own Work)

Brian Harvey – [Why SICP Matters](#)

Appendix on LC ( $\lambda c$ )

I decided to give a few pages of [Greg Michaelson's book](#) a try. It's a 2011 Dover-reprint of Michaelson's classic 1988 book.

I think this is a distraction, but it may be a required one. For a few days beginning 02/26/2024, whenever SICPJS shows up on my calendar, I will spend time on  $\lambda c$ . Let me reassess how far I

have come with the basics by 03/30/2024. I hope I complete at least chapters 2, 3, and 4 from the book.

26 February 2024

# Apoorv's Exercises

## Chapter 1

### [Exercise 1.1](#)

10  
12  
8  
3  
6  
3  
4  
19  
false  
4  
16  
6  
16

### [Exercise 1.2](#)

$(5 + 4 + (2 - (3 - (6 + 4 / 5)))) / (3 * (6 - 2) * (2 - 7))$

### [Exercise 1.3](#)

```
function largerTwoSquares(a, b, c) {  
  return (a > b)  
    ?  
    (b > c ? sum_of_squares(a, b) : sum_of_squares(a, c))  
    :  
    (a > c ? sum_of_squares(a, b) : sum_of_squares(b, c))  
}
```

## Exercise 1.4

The function adds  $a$  and  $|b|$ . It first checks whether or not  $b$  is nonnegative. If  $b$  is nonnegative, it calls the function `plus(a, b)`, and if  $b$  is negative, it calls the function `minus(a, b)`.

## Exercise 1.5

If the interpreter uses applicative-order evaluation, evaluating `test(0, p())` would first try to evaluate `p()`, which would lead to an infinite loop and cause the interpreter to exceed the maximum stack depth. If it uses normal-order evaluation, evaluating `test(0, p())` would first check whether `x === 0`, and since  $x$  is 0, it would immediately return 0 without evaluating `p()`.

## Exercise 1.6

Since the interpreter uses applicative-order evaluation, the interpreter will try to evaluate `sqrt_iter(improve(guess, x), x)` before passing it into the conditional function. This will result in infinite recursion.

Let us proceed with this problem in a step-by-step fashion. First, let us describe the result of the evaluation with a conditional expression.

```
function sqrt_iter(guess, x) {  
  return is_good_enough(guess, x)  
    ? guess  
    : sqrt_iter(improve(guess, x), x);  
}
```

In this case, when `sqrt_iter` is called, the interpreter will first evaluate the conditional expression. While evaluating the conditional, the interpreter will first try to evaluate `is_good_enough(guess, x)`. If it evaluates to true, the function `sqrt_iter` will return `guess`. Otherwise, the function will do a recursive call. The function works as designed.

Now let us describe what happens with the conditional function.

```
function conditional(predicate, then_clause, else_clause) {  
  return predicate ? then_clause : else_clause;  
}
```

```
function sqrt_iter(guess, x) {  
  return conditional(is_good_enough(guess, x),  
                    guess,  
                    sqrt_iter(improve(guess, x),  
                               x));  
}
```

```

        x));
}

```

When `sqrt_iter` is called, it will call the function `conditional`. But ***because the interpreter uses applicative-order evaluation*** and `conditional` is a function, the interpreter will attempt to evaluate `is_good_enough(guess, x)` and `sqrt_iter(improve(guess, x), x)` before calling the `conditional` function. So before the `conditional` can be evaluated, a (non terminating) recursive call is made. This will result in infinite recursion.

### Exercise 1.7

If you input a very small number into the `sqrt_iter` function, the difference between the square of the guess and the given number can still be less than 0.001 even if the square of the guess is double of the given number. For example, running `sqrt(5e-5)` returns 0.031781009679092864. The square of this number is  $1.01 \times 10^{-3}$ , which is around twenty times of  $5 \times 10^{-5}$ . However, the difference between  $1.01 \times 10^{-3}$  and  $5 \times 10^{-5}$  is less than 0.001. Furthermore, if you choose an even smaller number, the function returns a very similar number. Essentially, the function returns before guess is improved to be close enough.

A problem also occurs with very large numbers. The square of the guess might be within 0.1% of the actual number but the absolute difference could be greater than 0.001. This would cause the program to try to improve the guess. Since numbers are represented with limited precision, the program will end up improving the guess so little that the improvement is rounded to the guess. Essentially, the difference between the improved guess and the original guess is less than the minimum precision of the computer, so the guess and the improvement will be represented exactly the same by the computer. Hence, the function will encounter an infinite loop.

To fix these problems, we must check the relative precision of the guess, that is, we must check whether the square of the guess is within, say, 1% of the number given.

This improvement to the `is_good_enough` function checks whether the square of the guess is within error% of the given number.

```

function is_good_enough(guess, x, error) {
    return abs(square(guess) - x) < (x/(error*100));
}

```

### Exercise 1.8

Some small modifications can be made to the square root function to make a cube root function:

```

function abs(x) {

```

```

    return x >= 0 ? x : - x;
}

function square(x) {
    return x * x;
}

function cube(x) {
    return square(x) * x;
}

function is_good_enough(guess, x) {
    return abs(cube(guess) - x) < (x * 0.001);
}

function improve(guess, x) {
    return ((x/square(guess)) + (2 * guess))/3;
}

function cbrt_iter(guess, x) {
    return is_good_enough(guess, x)
        ? guess
        : cbrt_iter(improve(guess, x), x);
}

function cbrt(x) {
    return cbrt_iter(1, x);
}

```

The improvement to the function suggested in #7 is also implemented here. The function `cbrt(x)` will correctly print the cube root of `x`, in a way similar to the square root function but with a different improved guess formula.

This formula is derived with calculus. Suppose  $g$  is our guess for the cube root of  $n$ . Since we are trying to find the cube root, our function  $f(x) = \sqrt[3]{x}$ . Then,  $f'(x) = \frac{1}{3\sqrt[3]{x^2}}$ .

To get a new  $g$  closer to the value of  $f(n)$ , we use linear approximations. Since  $g$  is the guess of the cube root, it will be on the y-axis, and we introduce  $a$  such that  $f(a) = g$ . Consequently,  $a = f^{-1}(g) = g^3$ . Our current approximation is therefore  $(a, g)$ . To improve this approximation, we use  $f(x) \approx f'(a)(x - a) + f(a)$ , where  $f(x)$  is the improved approximation and  $f(a)$  is the

current approximation. Hence:

$$\begin{aligned}f(x) &= f'(g^3)(x - a) + g \\&= \frac{1}{3\sqrt[3]{(g^3)^2}}(x - g^3) + g \\&= \frac{x - g^3}{3g^2} + g \\&= \frac{x - g^3 + 3g^3}{3g^2} = \frac{x + 2g^3}{3g^2} \\&= \frac{x/g^2 + 2g}{3}\end{aligned}$$

### Exercise 1.9

Suppose we are trying `plus(4, 5)` with the first function. The operations will look like this:

```
plus(4, 5)
inc(plus(3, 5))
inc(inc(plus(2, 5)))
inc(inc(inc(plus(1, 5))))
inc(inc(inc(inc(plus(0, 5)))))
inc(inc(inc(inc(5))))
inc(inc(inc(6)))
inc(inc(7))
inc(8)
9
```

Since there is a chain of deferred incrementations, this process is recursive.

Now the second function may seem recursive because it calls itself. But in reality, it is just an iterative function with a seemingly recursive form. If we carry out `plus(4, 5)` with the second function, the operations will look like this:

```
plus(4, 5)
plus(3, 6)
plus(2, 7)
plus(1, 8)
plus(0, 9)
9
```

The second function is iterative because there are no deferred operations - `a` and `b` are updated

one after another. The **last** operation that the function does is a recursive call, so there are no deferred operations, and the function is just iterative.

### Exercise 1.10

```
A(1, 10)
A(0, A(1, 9))
A(0, A(0, A(1, 8)))
A(0, A(0, A(0, A(1, 7))))
...
A(0, A(0, A(0, A(0, A(0, A(0, A(0, A(0, A(1, 1))))))))))
A(0, A(0, A(0, A(0, A(0, A(0, A(0, A(0, A(0, 2))))))))))
A(0, A(0, A(0, A(0, A(0, A(0, A(0, A(0, 4))))))))))
A(0, A(0, A(0, A(0, A(0, A(0, A(0, 8))))))))))
...
1024
```

```
A(2, 4)
A(1, A(2, 3))
A(1, A(1, A(2, 2)))
A(1, A(1, A(1, A(2, 1))))
A(1, A(1, A(1, 2)))
A(1, A(1, A(0, A(1, 1))))
A(1, A(1, A(0, 2)))
A(1, A(1, 4))
A(1, A(0, A(1, 3)))
...
A(1, A(0, 8))
A(1, 16)
...
65536
```

```
A(3, 3)
A(2, A(3, 2))
A(2, A(2, A(3, 1)))a
A(2, A(2, 2))
A(2, A(1, A(2, 1)))
A(2, A(1, 2))
A(2, 4)
65536
```

It seems like the Ackermann function grows very rapidly. When the first parameter is 0, it seems to multiply the second parameter by 2. When the first parameter is 1, it raises the 2 to the second parameter - it performs repeated multiplication. When the first parameter is 2, it returns

$2^{2^{\dots}}$  with the number of 2's equal to the second parameter. That is, it seems to perform repeated exponentiation (also called tetration). Essentially, it seems that  $A(a, b)$  is repeated  $A(a-1, 2)$   $b$  times.

$$f(n) = A(0, n)$$

$$g(n) = A(1, n)$$

$$h(n) = A(2, n)$$

$$f(n) = 2n \rightarrow \text{multiplication}$$

$$g(n) = 2^n \rightarrow \text{exponentiation}$$

$$h(n) = 2 \uparrow\uparrow n \rightarrow \text{tetration}$$

## Exercise 1.11

[GitHub link](#)

Recursive implementation of  $f(n)$ :

```
function f(n) {
    return n < 3
    ? n
    : f(n - 1) + (2 * f(n - 2)) + (3 * f(n - 3));
}
```

Iterative implementation of  $f(n)$ :

```
function f_iter(n) {
    function f_int(r1, r2, r3, c) {
        return c === n
        ? r1
        : f_int(r1 + 2 * r2 + 3 * r3, r1, r2, c + 1);
    }
    return n < 3
    ? n
    : f_int(2, 1, 0, 2);
}
```



## Exercise 1.12

[GitHub link](#)

This function embodies a recursive process that calculates the number at row  $r$  and column  $c$  of Pascal's triangle. Both  $r$  and  $c$  are zero-indexed, so the first row or first column of a row would be  $r = 0$  and  $c = 0$  respectively.

```

function pascal(r, c) {
  return c < 0
    ? 0
    : c > r
    ? 0
    : r < 0
    ? 0
    : c === 0
    ? 1
    : c === r
    ? 1
    : pascal(r - 1, c - 1) + pascal(r - 1, c);
}

```

## Exercise 1.13

First, let us prove that  $Fib(n) = \frac{\phi^n - \psi^n}{\sqrt{5}}$ .

We construct a proof by induction. Assume  $Fib(r) = \frac{\phi^r - \psi^r}{\sqrt{5}}$ . Then:

$$\begin{aligned}
 Fib(r + 1) &= Fib(r) + Fib(r - 1) \\
 &= \frac{\phi^r - \psi^r}{\sqrt{5}} + \frac{\phi^{r-1} - \psi^{r-1}}{\sqrt{5}} \\
 &= \frac{\phi^r + \phi^{r-1} - (\psi^r + \psi^{r-1})}{\sqrt{5}} \\
 &= \frac{\phi^{r-1}(1 + \phi) - \psi^{r-1}(1 + \psi)}{\sqrt{5}} \\
 &= \frac{\phi^{r-1} \times \phi^2 \left(\frac{1+\phi}{\phi^2}\right) - \psi^{r-1} \times \psi^2 \left(\frac{1+\psi}{\psi^2}\right)}{\sqrt{5}}
 \end{aligned}$$

Recalling that  $\phi$  and  $\psi$  are the solutions to the equation  $x^2 - x - 1 = 0$ , we get  $\phi^2 = \phi + 1$

and  $\psi^2 = \psi + 1$ . Hence:

$$Fib(r + 1) = \frac{\phi^{r+1}(1) - \psi^{r+1}(1)}{\sqrt{5}} = \frac{\phi^{r+1} - \psi^{r+1}}{\sqrt{5}}$$

Additionally, we must provide base cases:

$$Fib(0) = 0$$

$$\frac{\phi^0 - \psi^0}{\sqrt{5}} = \frac{1-1}{\sqrt{5}} = 0$$

$$Fib(1) = 1$$

$$\frac{\phi^1 - \psi^1}{\sqrt{5}} = \frac{\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}}{\sqrt{5}} = \frac{\sqrt{5}}{\sqrt{5}} = 1$$

We can now confidently say that  $Fib(n) = \frac{\phi^n - \psi^n}{\sqrt{5}}$  for all  $n \geq 0$ . We still need to prove that

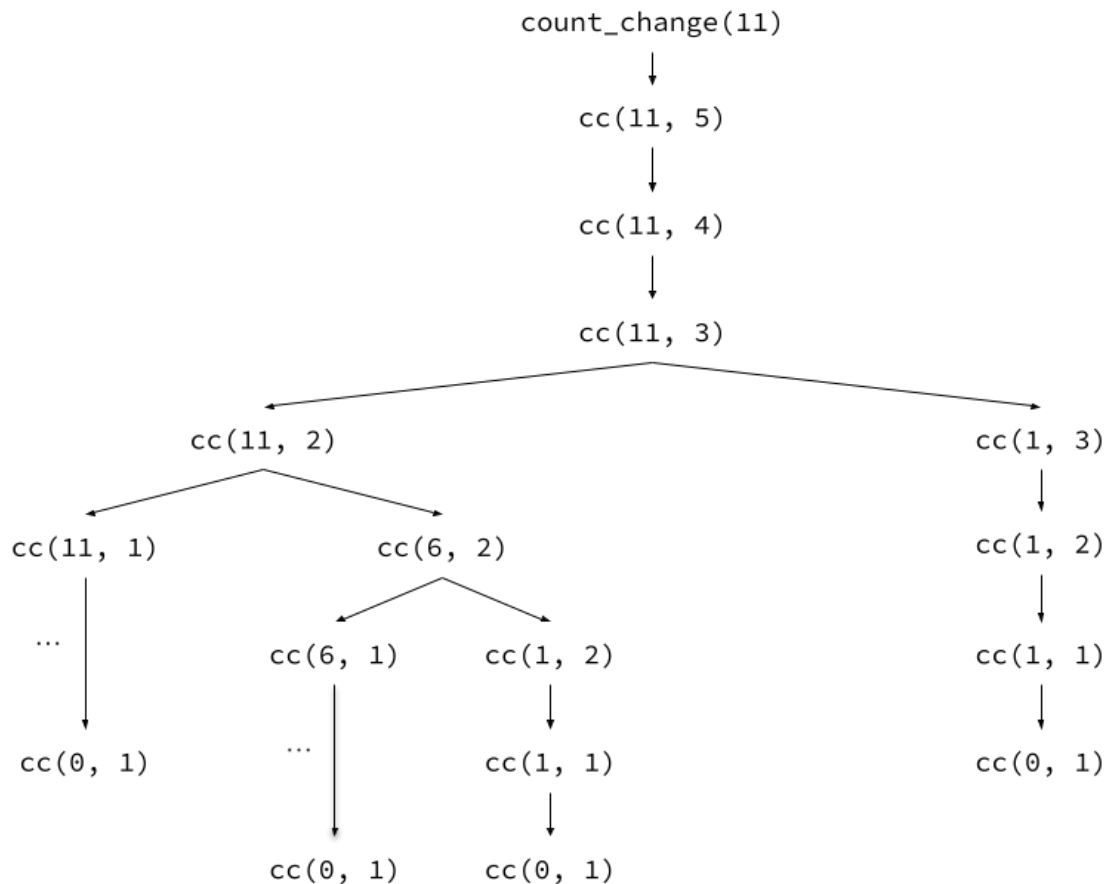
$Fib(n)$  is the closest integer to  $\frac{\phi^n}{\sqrt{5}}$ . In other words, we need to prove that  $\left| Fib(n) - \frac{\phi^n}{\sqrt{5}} \right| < 0.5$ .

$$\begin{aligned} \left| Fib(n) - \frac{\phi^n}{\sqrt{5}} \right| &= \left| \frac{\phi^n - \psi^n}{\sqrt{5}} - \frac{\phi^n}{\sqrt{5}} \right| \\ &= \left| \frac{-\psi^n}{\sqrt{5}} \right| \\ &= \left| \frac{-\frac{(1-\sqrt{5})^n}{2^n}}{\sqrt{5}} \right| \\ &= \left| \frac{-(1-\sqrt{5})^n}{2^n \sqrt{5}} \right| \\ &= \frac{|(1-\sqrt{5})^n|}{2^n \sqrt{5}} \\ &= \frac{|(1-\sqrt{5})^n|}{2^n \sqrt{5}} \end{aligned}$$

Need to complete.

## Exercise 1.14

Illustration of `count_change(11)` recursive call tree:



The time taken by the program increases with the total number of constant time operations performed. Each function call, except the base cases, recursively calls two functions, the results of which it sums. Hence, the number of additions (which are constant time operations) will be equal to the number of parent nodes in the tree. In the worst-case scenario, each parent node will have two child nodes. That is, the function call tree will be a complete binary tree.

The space taken by the program increases with the maximum depth of the stack memory. Since it is a recursive process, each in-flight (unreturned) function call must have a stack frame, and the maximum number of in-flight function calls will be equal to the maximum depth of the function call tree.

The longest path to a leaf in the tree will be the case where only the smallest denomination (call it  $s$ ) is used. The total number of calls in that path will be the number of times you reduce  $n$  by  $s$  plus the number of times you reduced the number of denominations by one. In other words,  $C(n, k, s) = n \div s + k$ , where  $C$  is the total number of calls and  $k$  is the number of denominations.

Now we are looking for the order of growth of space as  $n$  increases, which means  $k$  and  $s$  act as constants. Hence,  $S(n) = \Theta(n \div s + k) = \Theta(n)$ .

### Exercise 1.15

Here is the process generated by `sine(12.15)`:

```
sine(12.15)
p(sine(4.05))
p(p(sine(1.35)))
p(p(p(sine(0.45))))
p(p(p(p(sine(0.15)))))
p(p(p(p(p(sine(0.05)))))
p(p(p(p(p(0.05)))))
p(p(p(p(0.1505))))
p(p(p(0.46513545)))
... and so on.
```

In this process - `sine(12.15)` - the function `p` is applied five times.

Suppose  $T(a)$  is the number of steps required by `sine(a)`. Each call to `p` has five constant time operations (five steps). The number of steps will be five times the number of calls to `p`, which is equal to the number of function calls. Let us find the number of function calls, which we will call  $n$ :

$$\frac{a}{3^n} \leq 0.1 \text{ and } \frac{a}{3^{n-1}} \geq 0.1$$

$$3^n \geq 10a \text{ and } 3^{n-1} \leq 10a$$

$$n \geq \log_3(10a) \text{ and } n \leq \log_3(10a) + 1$$

Regardless of the exact value of  $n$ , it will be either the closest or second-closest integer to  $\log_3(10a)$ . So for our purposes, we can just say  $n = \log_3(10a)$ .

$$\log_3(10a) = \log_3(10) + \log_3(a) = \log_3(10) + \frac{1}{\log_{10}(3)} \cdot \log_{10} a$$

Since  $\log_3 10$  and  $1/\log_{10} 3$  are constants,  $n = \Theta(\log a)$ . Therefore,

$$T(a) = 5 * \Theta(\log a) = \Theta(\log a).$$

Suppose  $S(a)$  is the space required by `sine(a)`. Since there are deferred operations, each function call must be kept in the stack. Hence, the space required by `sine(a)` will be equal to the number of function calls. Using our result above,  $S(a) = \Theta(\log a)$ .

## Exercise 1.16

[GitHub link](#)

```
function fast_expt_iter(b, n) {
  function fast_expt_int(x, y, z) {
    return x === 0
      ? y
      : is_even(x)
        ? fast_expt_int(x / 2, y, square(z))
        : fast_expt_int(x - 1, y * z, z);
  }
  return fast_expt_int(n, 1, b);
}

function is_even(n) {
  return n % 2 === 0;
}

function square(n) {
  return n * n;
}
```

## Exercise 1.17

[GitHub link](#)

```
function fast_times(a, b) {
  return b === 0
    ? 0
    : is_even(b)
      ? double(fast_times(a, halve(b)))
      : a + fast_times(a, b - 1);
}

function is_even(n) {
  return n % 2 === 0;
}

function halve(n) {
  return n / 2;
}
```

```
function double(n) {
    return n + n;
}
```

## Exercise 1.18

[GitHub link](#)

```
function fast_times_iter(a, b) {
    function fast_times_int(x, y, z) {
        return z === 0
            ? y
            : is_even(z)
              ? fast_times_int(double(x), y, halve(z))
              : fast_times_int(x, x + y, z - 1);
    }
    return fast_times_int(a, 0, b);
}
```

```
function is_even(n) {
    return n % 2 === 0;
}
```

```
function halve(n) {
    return n / 2;
}
```

```
function double(n) {
    return n + n;
}
```

## Exercise 1.19

We should define  $T_{pq}$  somewhat mathematically. We can define it as a function which accepts a pair of arguments and returns a pair of arguments:

$$T_{pq}(a, b) = (bq + aq + ap, bp + aq)$$

Now, applying the transformation twice is equivalent to calling  $T_{pq}$  on  $T_{pq}$ , i.e.,  $T_{pq}(T_{pq}(a, b))$  or  $T_{pq} \circ T_{pq}(a, b)$ . This is essentially function composition.

$$\begin{aligned}
T_{pq}(T_{pq}(a, b)) &= T_{pq}(bq + aq + ap, bp + aq) \\
&= (q[bp + aq] + q[bq + aq + ap] + p[bq + aq + ap], p[bp + aq] + q[bq + aq + ap]) \\
&= (qbp + aq^2 + bq^2 + aq^2 + apq + pbq + apq + ap^2, bp^2 + apq + bq^2 + aq^2 + apq) \\
&= (2pbq + 2aq^2 + bq^2 + 2apq + ap^2, bp^2 + 2apq + bq^2 + aq^2) \\
&= (a(2q^2 + 2pq + p^2) + b(q^2 + 2pq), a(2pq + q^2) + b(q^2 + p^2)) - \text{Eq. 1}
\end{aligned}$$

We define  $T_{p'q'}(a, b) = T_{pq}(T_{pq}(a, b))$  and  $T_{p'q'}(a, b) = (bq' + aq' + ap', bp' + aq')$  - Eq. 2.

From the similarities between Eq. 1 and Eq. 2, we can find  $p'$  and  $q'$ :

$$\begin{aligned}
p' &= p^2 + q^2 \\
q' &= 2pq + q^2
\end{aligned}$$

If we plug  $p'$  and  $q'$  into Eq. 2, we should get Eq. 1.

Using the convention that  $T_{p'q'} = T_{pq}^2$ , we can now complete this function:

```

function fib(n) {
    return fib_iter(1, 0, 0, 1, n);
}

function fib_iter(a, b, p, q, count) {
    return count === 0
        ? b
        : is_even(count)
        ? fib_iter(a,
                    b,
                    square(p) + square(q),
                    square(q) + (2 * p * q),
                    count / 2)
        : fib_iter(b * q + a * q + a * p,
                    b * p + a * q,
                    p,
                    q,
                    count - 1);
}

function square(n) {
    return n * n;
}

```

### Exercise 1.20

gcd(206, 40)  
gcd(

### Exercise 1.21

### Exercise 1.22

This is the function search\_for\_primes ([GitHub link](#)):

```
function search_for_primes(start, end) {  
  function sfp_iter(curr) {  
    timed_prime_test(curr);  
    return curr + 2 > end  
      ? true  
      : sfp_iter(curr + 2);  
  }  
  return start % 2 === 0  
    ? sfp_iter(start + 1)  
    : sfp_iter(start);  
}
```

Here are the smallest prime numbers greater than 1000, 10000, 100000, and 1000000, along with their testing times:

Smallest primes greater than:	Prime	Testing time (ms)
$10^3$	1009	1
	1013	0
	1019	0
$10^4$	10007	0
	10009	1
	10037	1
$10^5$	100003	1
	100019	2
	100043	1



$10^6$	1000003	8
	1000033	6
	1000037	7

### [Exercise 1.23](#)

### [Exercise 1.24](#)

### [Exercise 1.25](#)

This function is essentially correct - it is doing exactly what we want `expmod` to do. However, exponentiation grows rapidly, and  $a^n$  is likely to exceed representation using 64 bits - the maximum number of bits for a number in JavaScript.

So we

### [Exercise 1.26](#)

### [Exercise 1.27](#)

### [Exercise 1.28](#)

### [Exercise 1.29](#)

The function `simpson` applies Simpson's rule to the function  $f$  to approximate the integral of  $f$  between  $a$  and  $b$ . ([GitHub link](#))

```
// applies simpson's rule to the function f
// finds the integral of f from a to b with n intervals
function simpson(f, a, b, n) {
  const h = (b - a) / n;
  function getnext(c) {
    return c === 0 || c === n
      ? f(a + c * h)
      : c % 2 === 1
      ? 4 * f(a + c * h)
      : 2 * f(a + c * h);
  }
```

```

    }
    return (h / 3) * sum(getnext, 0, x => x + 1, n);
}

```

```

simpson(cube, 0, 1, 100); // 0.24999999999999992
integral(cube, 0, 1, 100); // 0.249987500000000042

```

```

simpson(cube, 0, 1, 1000); // 0.25000000000000003
integral(cube, 0, 1, 1000); // 0.2499998750000001

```

### [Exercise 1.30](#)

```

function sum(term, a, next, b) {
  function iter(a, result) {
    return a > b
      ? result
      : iter(next(a), result + term(a));
  }
  return iter(a, 0);
}

```

### [Exercise 1.31](#)

[GitHub link](#)

The idea of the product function is the same as that of the sum function. The only differences are multiplying the term instead of adding and starting from one instead of zero (since multiplying by 1 has the same effect as adding 0).

```

function product_recur(term, a, next, b) {
  return a > b
    ? 1
    : term(a) * product_recur(term, next(a), next, b);
}

```

This is a recursive implementation. Similarly, here is an iterative implementation (analogous to the iterative implementation of sum):

```

function product_iter(term, a, next, b) {
  function iter(curr, total) {
    return curr > b
      ? total

```

```

        : iter(next(curr), total * term(curr));
    }
    return iter(a, 1);
}

```

The factorial of a number  $n$  is just the product of all the integers between 1 and  $n$ , inclusive. We use the identity function as `term`, and the increment function as `next` (both represented as arrow functions).

```

function factorial(n) {
    return product_iter(x => x, 1, x => x + 1, n);
}

```

Now we want to approximate  $\pi$  using the formula  $\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \cdots}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \cdots}$ . Our product function will come in handy here.

Say we want to continue this approximation for  $n$  terms. There are two ways we can go about this. The first way is to calculate the product of  $n$  integers in both the numerator and denominator, and then divide the two numbers.

The `term` function in the product of the numerator would have to map  $n$  natural numbers to the first  $n$  numbers in the sequence 2, 4, 4, 6, 6, 8, .... The `term` function in the product of the denominator would have to map  $n$  natural numbers to the first  $n$  numbers in the sequence 3, 3, 5, 5, 7, 7, ....

Since the numbers in the sequences come in pairs, the `term` functions must take into account the remainder of the number when divided by 2 ( $x \bmod 2$  or  $x \% 2$ ). After a little bit of trial and error, we discover  $n(x) = x + 2 + (x \% 2)$  which returns 2 when  $x$  is 0, 4 when  $x$  is 1 or 2, 6 when  $x$  is 3 or 4, and so on. Similarly, we find  $d(x) = x + 3 - (x \% 2)$  which returns 3 when  $x$  is 0 or 1, 5 when  $x$  is 2 or 3, 7 when  $x$  is 4 or 5, and so on. Now that we have the `term` functions, we can construct the calls to `product`.

```

function approxpi1(n) {
    return product_iter(
        x => x + 2 + (x \% 2),
        0,
        x => x + 1,
    n) / product_iter(
        x => x + 3 - (x \% 2),
        0,
        x => x + 1,
    n);
}

```

```

    n) * 4;
}

```

`approxpi1(150)` gives us 3.131342841085697 - not a bad approximation, but not a great one. But if you try to get a more precise approximation by including more terms, you quickly run into a problem. Even a small number such as `approxpi1(200)` spits out NaN, which is not surprising. Multiplying so many numbers which continue to increase is certain to exceed even 1024-bit representation. This brings us to the second method for approximating  $\pi$ . Instead of first calculating the product and then dividing, first calculate the  $n$ th fraction and then calculate the product of those fractions. That is, instead of  $\frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \cdots}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \cdots}$ , calculate  $\frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \frac{8}{7} \cdots$ . We only need one product call - to construct it, we can use the expressions derived for the term functions above.

```

function approxpi2(n) {
  return product_iter(
    x => (x + 2 + (x % 2)) / (x + 3 - (x % 2)),
    0,
    x => x + 1,
    n) * 4;
}

```

This gives the exact same results as `approxpi1`, except it can exceed  $n=150$  - `approxpi2(200)` returns 3.133864293497815. In fact, `approxpi2(300000)` gives 3.1415874176583385 - accurate to 4 decimal places, much more accurate than the 1 decimal place possible with `approxpi1`.

However, we run into a different problem now. Although `approxpi2(300000)` runs in under a second, as you increase  $n$  further, it starts to take noticeable time. `approxpi2(3000000)` takes nearly ten seconds, and it only grows from there. Ultimately, not every program can be perfect - each will have limitations. We therefore try to approximate perfection as best as possible.

## [Exercise 1.32](#)

[GitHub link](#) for `accumulate_recursive`, `accumulate_iterative`, `product`, and `sum` functions.

```

function accumulate_recur(combiner, null_value, term, a, next, b) {
  return a > b
    ? null_value
    : combiner(term(a), accumulate_recur(combiner, null_value, term,
next(a), next, b));
}

```

```

}

function accumulate_iter(combiner, null_value, term, a, next, b) {
  function acc(curr, total) {
    return curr > b
      ? total
      : acc(next(curr), combiner(total, term(curr)));
  }
  return acc(a, null_value);
}

function product(term, a, next, b) {
  return accumulate_iter((x, y) => x * y, 1, term, a, next, b);
}

function sum(term, a, next, b) {
  return accumulate_iter((x, y) => x + y, 0, term, a, next, b);
}

```

### [Exercise 1.33](#)

This is a slightly more generalized version of `accumulate`. It is not much of an improvement - it just adds a filter on the numbers that the function `accumulate` checks. This is usually only useful if the numbers to be added follow a vague pattern but are not evenly spaced, such as prime numbers (all odd numbers but not evenly spaced).

```

function filtered_accumulate(combiner, null_value, term, a, next, b, filter) {
  return a > b
    ? null_value
    : filter(a)
      ? combiner(term(a), filtered_accumulate(combiner, null_value, term, next(a),
next, b, filter))
      : filtered_accumulate(combiner, null_value, term, next(a), next, b, filter);
}

```

This function can be recursive or iterative, just like `accumulate`.

```

function filtered_accumulate_iter(combiner, null_value, term, a, next, b, filter) {
  function filter_iter(curr, total) {
    return curr > b
      ? total
      : filter(curr)
        ? filter_iter(next(curr), combiner(total, term(curr)))
        : filter_iter(next(curr), total);
  }
}

```

```

    }
    return filter_iter(a, null_value);
}

```

To find the sum of the squares of the prime numbers between  $a$  and  $b$ :

```

function primes_square_sum(a, b) {
    return filtered_accumulate_iter(
        (x, y) => (x + y),
        0,
        x => square(x),
        a,
        x => x + 1,
        b,
        is_prime);
}

```

Similarly, for the product of all the positive integers less than  $n$  that are relatively prime to  $n$ :

```

function coprime_product(n) {
    function is_coprime(i) {
        return gcd(n, i) === 1;
    }
    return filtered_accumulate_iter(
        (x, y) => x * y,
        1,
        x => x,
        1,
        x => x + 1,
        n,
        is_coprime);
}

```

[GitHub link](#)

## [Exercise 1.34](#)

We can use the substitution model to

## Chapter 2

## [Exercise 2.7](#)

## Miscellaneous

- Write an iterative process `fib(n)` that finds the  $n$ th fibonacci number, using the Source §1 sublanguage. (Source §1 has no loops, so the function must be recursive.)

```
■ function fib(n) {  
    function fib_iter(a, b, c) {  
        return c == n  
            ?  
            a  
            :  
            fib_iter(b, a+b, c+1);  
    }  
    return fib_iter(1, 1, 1);  
■  
}  
  
fib(6); // => 8
```

## Video Lectures

## Our Notes

29 November 2022

Food for thought: In your favorite programming language, write a program that, when run, will print out its own source code.

25 November 2022

We read the [1984-foreword](#) written by Alan J. Perlis to the original SICP book. This foreword contains some of the best words ever written about programming languages. Good stuff. Apoorv had a lot of fun. He was irritable when we started (because of college admission apps), but then as Perlis's words were read, he got hooked.

Kedar's Impressions of Perlis's Foreword

This is beautiful writing. Perlis has succinctly summarized how

Home work:

- 1) Read Guy Steele's [foreword](#) written in 2021 (almost 40 years after the original foreword) and compare the two.
- 2) Make progress on the Lisp paper. Annotate it as much as possible. You can [annotate](#) your own way.

### Concerns

Apoorv needs to spend more time on this. Make progress on the AI's or home work.

### 16 November 2022

We started reading a paper about Lisp written by Abelson and Sussman. The paper is, like most of SICP, more about the programming process and some principles underlying all programming languages, than about any particular programming language.

### Home Work:

- 1) Read and [annotate the paper](#). **Deadline: 07 December ~~30 November~~ 2022.**
- 2) Read Brian Harvey's [article](#).