1.2 - Procedures and the Processes They Generate

- Must be able to visualize how the program / procedure will run before executing it
    - Then, can change according to it
- Procedure - a pattern for *local evolution* of a computational process
    - Shows how each stage of process built upon previous stage
- *Global behavior* - eventual goal to be able to describe

1.2.1 - Linear Recursion and Iteration

- Linear recursive process for factorials

```
(define (factorials n)
  (if (= n 1)
      1
      (* n (factorials (- n 1)))))
```

- Linear iterative process for factorials

```
(define (fact-iter product counter n)
  (if (> counter n)
      product
      (fact-iter (* product counter)
                 (+ counter 1)
                 n)))

(define (fact-v2 n)
  (fact-iter 1 1 n))
```

- Substitution model - good way to visualize how each procedure runs
- *Linear Recursive Process*
    - *Recursive Process* - characterized by a chain of *deferred operations* (operations stored in memory to be evaluated later)
    - *Linear* - the chain of deferred operations (/ amount of info needed to be stored in memory) grows proportionally to the parameter (in the case of the factorials, to *n*)
- *Linear Iterative Process*
    - *Iterative Process* - characterized by a fixed number of *state variables* (variables which contain enough info to carry out future behaviors necessary by the process)
        - Has a fixed rule to describe how state variables change as process evolves
        - Has optional end test to determine if process should terminate

    - *Linear*- number of steps required to compute the process grows (is proportional) linearly with the formal parameter (in case of factorials, *n*)

Linear Recursive Process	**Linear Iterative Process**
● When ran, has “hidden” info stored in system/interpreter in relation to the chain of deferred operations ● Longer chain = more info stored	● System only maintains state variables ● To stop and resume process, only the state variables need to be stored

- Recursive **procedure** - describes how syntactically the procedure definition refers, directly or indirectly, to itself
- Recursive **process** - describes how a process evolves in some sort of a recursive manner

- Most programming languages consume memory in proportion to the recursive procedure calls
    - Thus, process not truly iterative
    - Uses special looping constructs to perform iteration (e.g. for, while, etc.)
- Solution in Scheme
    - *Tail Recursion* - iterative process always stays within a constant space (within memory)

Exercise 1.9

```
(define (+ a b)
  (if (= a 0)
      b
      (inc (+ (dec a) b))))
```

Recursive Process

```
(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))
```

Iterative Process

Exercise 1.10

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1)
                 (A x (- y 1))))))
```

(A 1 10) = 1024

(A 2 4) = 65536

(A 3 3) = 65536

(define (f n) (A 0 n))
= 2n

(define (g n) (A 1 n))
$=2^n$

(define (h n) (A 2 n))
$=2^{\text{^}2\text{^}2\text{^}2\ldots}$ (n number of times)

1.2.2 - Tree Recursion

- Fibonacci process

```
(define (fib n)
        (cond ((= n 0) 0)
              ((= n 1) 1)
              (else (+ (fib (- n 1))
                       (fib (- n 2)))
```

- In LISP, this demonstrates tree recursion
    - However not every efficient, since much of recursive calls are repeated
        - i.e. fib(1) and fib (0)
    - Redundant
    - Number of fib(1) and fib(0) = fib(n+1)
    - Space required grows linearly, as only ondes above current evaluations need to be kept track of
        - Proportional to the max depth of tree
    - Steps required grows exponentially
- About fibonacci series
    - Grows exponentially

    - fib(n) = closest integer to $\phi^n/\sqrt{5}$
    - $\phi = (1+\sqrt{5})/2 \approx 1.6180$ and
    - $\phi^2 = \phi + 1$
- Iterative process for fibonacci #'s

```
(define (fib-iter a b count)
  (if (= count 0)
      a
      (fib-iter b (+ a b) (- count 1))))

(define (fibv2 n)
  (fib-iter 0 1 n))
```

- Benefits of tree recursion
    - Allows for easy prototyping of processes
    - Powerful tool when dealing with hierarchically structured data

Example: Counting change

- Problem: to calculate the total number of ways to make *an amount* of money in change with pennies, nickels, dimes, quarters, and half dollars

- Solution:
    - Basic assumptions made
        - If *a* = 0, then 1 way to make change
        - If *a* < *0,* then 0 ways to make change
        - If *n* = *0*, then 0 ways to make change
    - Recursively, can be modeled by two statements
        - If only using one denomination of coins, then one group could be just those coins
        - The 2nd group would start with one use of the chosen denomination of coins
            - The process would then repeat from the first statement, with *a-d*, *d* being the chosen denomination
            - This process would recursively call itself until every combination is exhausted
                - Which, at the end, is tallied up since amount would eventually equal 0
        - The process is repeated starting at a different denomination of coins until every combination is resolved
    - Entire process recursive, and thus uses up a lot of memory

```
(define (count-change amount)
  (cc amount 5))

(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                     (- kinds-of-coins 1))
                 (cc (- amount
                        (first-denom kinds-of-coins))
                     kinds-of-coins)))))

(define (first-denom kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))
```

Exercise 1.11

- Solution

Recursive Process

```
(define (f n)
  (cond ((< n 3) n)
        ((>= n 3) (+ (f (- n 1))
                     (* 2
                       (f (- n 2)))
                     (* 3
                       (f (- n 3)))))))
```

Iterative Process

```
(define (f-iter n a b c)
   (if (= 3 n) (+ (* a 2)
                  (* b 1)
                  (* c 0))
               (f-iter (- n 1)
                       (+ a b)
                       (+ (* 2 a) c)
                       (* 3 a))))
```

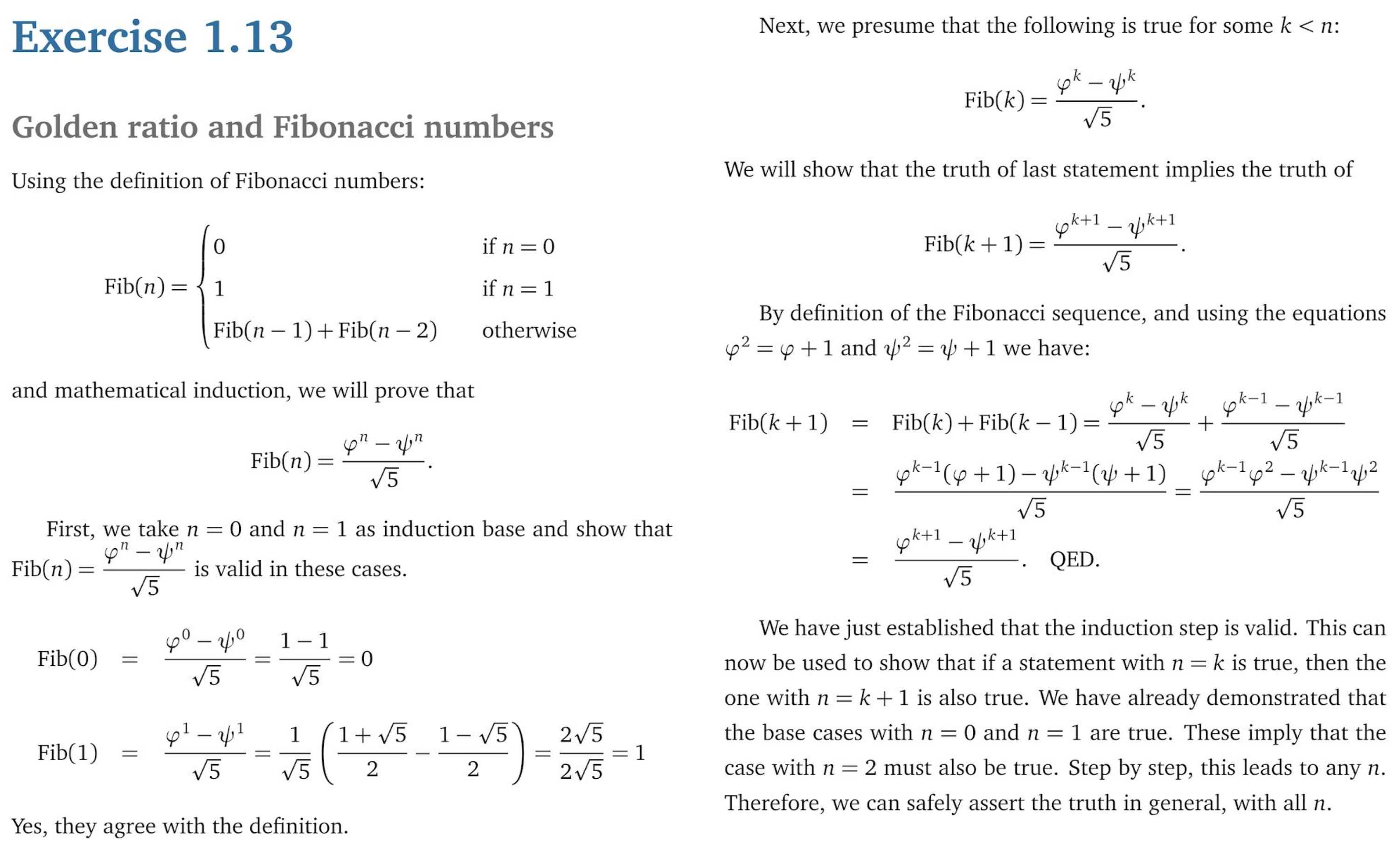
```
(define (fv2 n)
  (if (< n 3)
      n
      (f-iter n 1 2 3)))
```

Exercise 1.12

```
#lang sicp

(define (repeat word count)
  (cond ((< count 0) false)
        ((> count 0) (and (display word)
                          (display " ")
                          (repeat word (- count 1))))))

(define (pascal-elem n m)
  (cond ((or (> n m) (< n 0) (< m 0)) 0)
        ((or (= n m) (= n 0)) 1)
        (else (+ (pascal-elem (- n 1) (- m 1))
                 (pascal-elem n (- m 1))))))

(define (get-row-raw row column count)
  (cond ((< column 0) (display ""))
        ((> column row) (display "\n"))
        (else (let ((elem (pascal-elem count row)))
                (and (display elem)
                     (display " ")
                     (get-row-raw row (- column 1) (+ 1 count)))))))

(define (get-row row column)
  (get-row-raw row column 0))
```

Exercise 1.13

No idea, but searched up proof



Exercise 1.13

Golden ratio and Fibonacci numbers

Using the definition of Fibonacci numbers:

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

and mathematical induction, we will prove that

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

First, we take $n = 0$ and $n = 1$ as induction base and show that $\mathrm{Fib}(n) = \frac{\varphi^n - \psi^n}{\sqrt{5}}$ is valid in these cases.

$$\begin{aligned} \mathrm{Fib}(0) &= \frac{\varphi^0 - \psi^0}{\sqrt{5}} = \frac{1-1}{\sqrt{5}} = 0 \\ \mathrm{Fib}(1) &= \frac{\varphi^1 - \psi^1}{\sqrt{5}} = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}\right) = \frac{2\sqrt{5}}{2\sqrt{5}} = 1 \end{aligned}$$

Yes, they agree with the definition.

Next, we presume that the following is true for some $k < n$:

$$\mathrm{Fib}(k) = \frac{\varphi^k - \psi^k}{\sqrt{5}}.$$

We will show that the truth of last statement implies the truth of

$$\mathrm{Fib}(k+1) = \frac{\varphi^{k+1} - \psi^{k+1}}{\sqrt{5}}.$$

By definition of the Fibonacci sequence, and using the equations $\varphi^2 = \varphi + 1$ and $\psi^2 = \psi + 1$ we have:

$$\begin{aligned} \mathrm{Fib}(k+1) &= \mathrm{Fib}(k) + \mathrm{Fib}(k-1) = \frac{\varphi^k - \psi^k}{\sqrt{5}} + \frac{\varphi^{k-1} - \psi^{k-1}}{\sqrt{5}} \\ &= \frac{\varphi^{k-1}(\varphi+1) - \psi^{k-1}(\psi+1)}{\sqrt{5}} = \frac{\varphi^{k-1}\varphi^2 - \psi^{k-1}\psi^2}{\sqrt{5}} \\ &= \frac{\varphi^{k+1} - \psi^{k+1}}{\sqrt{5}}. \quad \text{QED.} \end{aligned}$$

We have just established that the induction step is valid. This can now be used to show that if a statement with $n = k$ is true, then the one with $n = k + 1$ is also true. We have already demonstrated that the base cases with $n = 0$ and $n = 1$ are true. These imply that the case with $n = 2$ must also be true. Step by step, this leads to any n. Therefore, we can safely assert the truth in general, with all n.

1.2.3 - Orders of Growth

- Processes differ in rates which computational resources (memory) are consumed
    - *Order of Growth* - gross measure of resources required by a process as inputs become larger
        - Offers a crude estimation, but useful in predicting general behavior
        - $\theta(n)$ is linear - if n doubled, so will space
        - $\theta(n^2)$ is exponential - if n doubled, space increases by constant multiplied rate - exponential
        - Logarithmic order of growths increases space by a constant amount when problem size (n) is doubled
    - ***n*** = parameter to measure the size of problem
        - e.g. if square root, how many decimals of accuracy
        - If matrix multiplication, how many rows
    - ***R(n)*** = measures number of internal storage registers used
    - R(n) = $\theta(f(n))$ -[the order of growth which R(n) has when positive constants k1 and k2 independent of n such that $k1f(n) \le R(n) \le k2f(n)$ for sufficient large values of n
- e.g.
    - Linear recursive process - factorials
        - Steps and space grow proportionally to input n
            - Thus, grow proportional to $\theta(n)$
    - Iterative process -factorials
        - Steps grows proportionally with $\theta(n)$
        - Space is constant - $\theta(1)$

Exercise 1.14

Order of growth - space

- $\theta(n)$ is the order of growth of space. The space required is proportional to the amount of data input, and as the depth of the tree is at its greatest n, then the order of growth for space is $\theta(n)$

Time complexity

- $\theta(n^5)$
- Explanation
  http://www.billthelizard.com/2009/12/sicp-exercise-114-counting-change.html

Exercise 1.15

a. 12.15 needs to be divided by 3 every time p is applied
    i. Thus, 3^x=121.5 when solved for x would get the value for which 12.15 would need to be divided to get less than or equal to 0.1
    ii. x=4.37→ 5 times

b. The time complexity and the growth of space is O($\log_3 a$) as the angle is constantly divided by 3.

1.2.4 - Exponentiation

- Calculating exponents as a algorithm
  - Base cases -
    - $b^n=b*b^{n-1}$
    - $b^0=1$
  - Recursive process

```
(define (expt b n)
        (if (= n 0)
        1
        (* b (expt b (-n 1))
```

  - Linear recursive process

```
(define (expt-iter b counter product)
        (if (= count 0))
                product
        (expt-iter (b
                     (- counter 1)
                     (* b product)))
```

  - Overall time complexity
    - Recursive - requires $\theta(n)$ steps and space, as solely depends on the exponent to determine how many multiplications
    - Iterative - requires $\theta(n)$ steps, as multiplication still depends on exponent
      - But, $\theta(1)$ space as it is iterative
- To reduce steps and space
  - Use successive squaring for calculating exponentiation
    - Base conditions
    - $b^n = (b^{n/2})^2$ if n is even
    - $b^n = b*b^{n-1}$ if n is odd
  - Function

```
(define (fast-expt b n)
        (cond   ((= n 0) 1)
                ((even n)  (square (fast-expt b (/ n 2))))
                (else (* b (fast-expt b (- n 1))))))

(define (square n)
        (* n n))

(define (even)
                          (= (remainder n 2) 0))
```

- Time complexity
    - $\theta(log(n))$, as it's constantly divided by 2
    - Space is $\theta(n)$, as the steps still depends on number of recursions

Exercise 1.16

```
(define (fast-expt-iter b n a)
  (cond ((= n 1) a)
        ((even n) (fast-expt-iter b
                                  (/ n 2)
                                  (* a (square b))))
        (else (fast-expt-iter (square b)
                              (- n 1)
                              a))))

(define (fast-expt b n)
  (fast-expt-iter b n 1))


(define (square x)
  (* x x))

(define (even x)
  (= (remainder x 2) 0))
```

Exercise 1.17 / 1.18

- Recursive process

```
(define (mult a b)
  (if (or (= a 0) (= b 0))
      0
      (+ a (mult a (- b 1)))))
```

- Iterative process

```
(define (fast-mult a b)
  (mult-iter a b 0))
```

```
(define (mult-iter a b n)
  (cond ((= b 0) 0)
        ((= a 0) n)
        ((even a) (mult-iter (half a)
                             (double b)
                              n))
        (else (mult-iter (- a 1)
                         b
                         (+ n b)))
              ))

(define (double x)
  (+ x x))

(define (half x)
  (/ x 2))

(define (even x)
  (= (remainder x 2) 0))
```

Exercise 1.19

don’t know the math, searched up

Exercise 1.19

Derivation of the new transformation

Here is the general transformation on a pair of integers:

$$T_{pq}(a,b) = \begin{cases} a \leftarrow a(p+q) + bq \\ b \leftarrow bp + aq \end{cases}$$

When $p = 0$ and $q = 1$, this reduces to the familiar case generating consecutive *Fibonacci* numbers if used with the seed pair $(1,0)$:

$$T = T_{01}(a,b) = \begin{cases} a \leftarrow a + b \\ b \leftarrow a \end{cases}$$

We get a new transformation $T_{p'q'}$ by applying the transformation T_{pq} twice:

$$T_{p'q'} = T_{pq}(T_{pq}) = T_{pq} \cdot T_{pq} = T_{pq}^2$$

This means that we replace every a with $a(p+q)+bq$ and every b with $bp+aq$ in the first formula of T_{pq}:

$$T_{pq}^2(a,b) = \begin{cases} a \leftarrow (a(p+q)+bq)(p+q) + (bp+aq)q \\ b \leftarrow (bp+aq)p + (a(p+q)+bq)q \end{cases}$$

After multiplying out and rearranging, we get:

$$T_{pq}^2(a,b) = \begin{cases} a \leftarrow a(p^2+q^2+2pq+q^2) + b(2pq+q^2) \\ b \leftarrow b(p^2+q^2) + a(2pq+q^2) \end{cases}$$

By comparing this to the first formula, we see that:

$$\begin{cases} p' = p^2 + q^2 \\ q' = 2pq + q^2 \end{cases}$$

Now we are ready to use this result to complete the program.

Program

```
(define (square x) (* x x))

(define (fib n)
  (fib-iter 1 0 0 1 n))

(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a
                   b
                   (+ (square p) (square q)) , p’
                   (+ (* 2 p q)  (square q)) , q’
                   (/ count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))
```

1.2.5 - Greatest Common Divisor

- Greatest Common Divisor (GCD)
    - Greatest number that is divisible by a set of integers with no remainder
    - Usually - would factor each integer, search for largest common factor
    - More clever algorithm - Euclid's Algorithm
        - Observation - if two integer a, b, with remainder r when a/b ; common divisors of a and b is same as common divisor of b and r
        - e.g. GCD(a, b) → GCD(b, r)
            - Repeats until remainder is 0, which would mean the GCD is calculated at the a position
        - Overall, iterative process
            - Time complexity = θ(log n)
- 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

Exercise 1.20

- Remainder applied 18 times - normal order
- Remainder applied 4 times - applicative order

1.2.6 - Testing for Primality

- Method 1- searching for divisors
    - Algorithm finds number's divisors greater than 1
        - Increments the test-divisor by 1 each iteration
    - Principle - find divisor end test based on how random argument *n* can not be prime if it has a divisor <= it's square root
        - i.e. n is prime when the square of the smallest test divisor is found be > *n* and also not be divisible into *n*
        - (prime n) compares with n, if equal, then the smallest divisor is the number itself, thus prime
    - Time complexity - O(√n)
        - Only numbers between 1 and √n are tested due to how any numbers above that means n is prime
            - Therefore, time complexity is √n

```
(define (prime n)
  (= n (smallest-divisor n)))
```

```
(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divide? test-divisor n) test-divisor)
        (else (find-divisor n (+ 1 test-divisor)))))

(define (divide? test-divisor n)
  (= (remainder n test-divisor) 0))

(define (square n)
  (* n n))
```

- Method 2- The Fermat Test
    - Time complexity of algorithm = $\theta(log(n))$
    - **Fermat's Little Theorem**
        - If *n* is a prime number and *a* is any positive integer less than *n*, then *a* raised to the *n*th power is congruent to *a* modulo *n*
            - i.e. the a modulo n is the same thing as the remainder when *a* raised to the *n* power is divided by *n*
            - $a^n \equiv a$ mod n
        - Modulo - remainder of *a* when divided by *n*
        - Test performed repeatedly for greater accuracy
    - Fermat's test - only one so far with almost certain success rate
        - Success rate can even be changed to desired accuracy

Algorithm for fermat's test for primality

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m))
                              m))
        (else
         (remainder (* base (expmod base (- exp 1) m))
                     m)
       )))

(define (square x)
  (* x x))

(define (even? y)
 (= (remainder y 2) 0))

(define (fermat-test n)
  (define (try a)
    (= (expmod a n n) a))
  (try (+ 1 (random (- n 1)))))

(define (prime-test n times)
  (cond ((= times 0) true)
        ((fermat-test n) (prime-test n (- times 1)))
        (else false)))
```

- Method 3 - probabilistic methods
    - The ability to create and augment primality algorithms so that the probability of failure is as small as needed
        - Chance of error usually becomes arbitrarily small

    - Method 3.1 -Miller-Rabin test
    - Method 3.2 - RSA algorithm

Exercise 1.21

```
(define (smallest-divisor n)
  (find-divisor n 2))
(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divide? n test-divisor) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))
(define (square n)
  (* n n))
(define (divide? a b)
  (= 0 (remainder a b)))

(smallest-divisor 199)
            → 199

(smallest-divisor 1999)
            → 1999

(smallest-divisor 19999)
            → 7
```

Exercise 1.22

```
(define (prime? n)
  (define (square x)
    (* x x))
  (define (divide? a b)
    (= 0 (remainder a b)))
  (define (find-divisor n test-divisor)
    (cond ((> (square test-divisor) n) n)
          ((divide? n test-divisor) test-divisor)
          (else (find-divisor n (+ test-divisor 1)))))
  (define (smallest-divisor n)
    (find-divisor n 2))
  (= (smallest-divisor n) n))


(define (timed-prime-test n)
  (start-prime-test n (runtime)))
(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime n (- (runtime) start-time))))
(define (report-prime n elapsed-time)
  (newline)
  (display n)
  (display " *** ")
  (display elapsed-time))

(define (even? n)
  (= 0 (remainder n 2)))

(define (search-for-primes t1 tn)
  (define (search-iter t tn)
    (if (<= t tn) (timed-prime-test t))
    (if (<= t tn) (search-iter (+ t 2) tn)))
  (search-iter (if (even? t1) (+ 1 t1) t1)
               (if (even? tn) (- tn 1) tn)))
```

Above 1000

- 1009
- 1013
- 1019

Above 10000

- 10007
- 10009
- 10037

Above 100000

- 100003
- 100019
- 100043

Above 1000000

- 1000003
- 1000033
- 1000037

Computer is too fast to have any noticeable change in runtime. This was tried with larger values

(search-for-primes 1000000000 1000000050)

- 1000000007 *** 1995
- 1000000009 *** 2022
- 1000000021 *** 1994
- 1000000033 *** 1999

Runtime increase from last

> (search-for-primes 10000000000 10000000050)

- 10000000019 *** 4985
- 10000000033 *** 4984

Runtime increase from last - 2.5

> (search-for-primes 100000000000 100000000050)

- 100000000003 *** 16952
- 100000000019 *** 15957

Runtime increase from last - 3.2

(search-for-primes 1000000000000 1000000000050)

- 1000000000039 *** 53856

Runtime increase from last - 3.375

```
(search-for-primes 10000000000000 10000000000050)
```

- 10000000000037 *** 179497

Runtime increase from last - 3.33

Value (input) increases by a factor of 10, runtime increases by around 3.25 each time. Time complexity is $\sqrt{n}$

$\sqrt{10} = 3.162$

Exercise 1.23

```
(define (prime? n)
  (define (square x)
    (* x x))
  (define (divide? a b)
    (= 0 (remainder a b)))
  (define (find-divisor n test-divisor)
    (cond ((> (square test-divisor) n) n)
          ((divide? n test-divisor) test-divisor)
          (else (find-divisor n (next test-divisor)))))
  (define (smallest-divisor n)
    (find-divisor n 2))
  (= (smallest-divisor n) n))

(define (next test-divisor)
  (cond ((= test-divisor 2) 3)
        (else (+ 2 test-divisor))))
```

```
(search-for-primes 1000000000 1000000050)
```

- 1000000007 *** 998
- 1000000009 *** 998
- 1000000021 *** 997
- 1000000033 *** 997

```
> (search-for-primes 10000000000 10000000050)
```

- 10000000019 *** 2992
- 10000000033 *** 2992

```
> (search-for-primes 100000000000 100000000050)
```

- 100000000003 *** 9972
- 100000000019 *** 9974

Runtime is twice as fast as before

Exercise 1.24

```
;primality test
(define (prime? n)
  (define (expmod base exp m)
    (cond ((= exp 0) 1)
          ((even? exp) (remainder (square (expmod base (/ exp 2) m))
                                  m))
          (else (remainder (* base
                              (expmod base (- exp 1) m))
                           m))))
  (define (fermat-test n)
    (define (try a)
      (= (expmod a n n) a))
  (if (= n 1)
      (try (+ 1 (random n)))
      (try (+ 1 (random (- n 1))))))
  (fermat-test n)
  )



;run time
(define (timed-prime-test n)
  (start-prime-test n (runtime)))

(define (start-prime-test n start-time)
  (if (prime? n)
      (report-prime n (- (runtime) start-time))))

(define (report-prime n elapsed-time)
  (newline)
  (display n)
  (display " *** ")
  (display elapsed-time))


;search for primes in range
(define (search-for-primes t1 tn)
  (define (search-iter t tn)
    (if (<= t tn) (timed-prime-test t))
    (if (<= t tn) (search-iter (+ t 2) tn)))
  (search-iter (if (even? t1) (+ 1 t1) t1)
               (if (even? tn) (- tn 1) tn)))
```

(search-for-primes 1000000000 1000000050)

- 1000000007 *** 0
- 1000000009 *** 0
- 1000000021 *** 0
- 1000000033 *** 0

> (search-for-primes 10000000000 10000000050)

- 10000000019 *** 0
- 10000000033 *** 0

> (search-for-primes 100000000000 100000000050)

- 100000000003 *** 0
- 100000000019 *** 0

Discrepancy -- system is too fast to appreciate the runtime

- Modern computers not able to produce noticeable runtime differences with iterative procedures and small inputs
- Thus, this just has to be assumed to have an order of growth of ϴ(log n)

Exercise 1.25

```
(define (fast-exp base n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-exp base (/ n 2))))
        (else (* base (fast-exp base (- n 1))))))

(define (square x)
  (* x x))

(define (even x)
  (= (remainder x 2) 0))

(define (expmod base exp m)
  (remainder (fast-exp base exp) m))
```

- This works. The fast-exp simply separates the fast-exp into a seperate definition

- The procedure remains the same but looks cleaner
- Might be insignificantly slowly as need to call a separate procedure
- Overall, would work for the fast-prime test

Exercise 1.26

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (* (expmod base (/ exp 2) m)
                       (expmod base (/ exp 2) m))
                    m))
        (else
         (remainder (* base (expmod base (- exp 1) m))
                    m))))
```

- This generates a tree recursion, which is as deep as the input m
- Thus, it has time complexity ϴ(n)

Exercise 1.27

```
;expmod
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m))
                            m))
        (else
         (remainder (* base (expmod base (- exp 1) m))
                    m)
       )))

;fermat procedure
(define (fermat-test n counter)
  (define (try a)
    (= (expmod a n n) a))
  (try counter))


;primality test
(define (prime-test-iter n counter)
  (cond ((= counter 0) true)
        ((fermat-test n counter) (prime-test-iter n (- counter
                                                       1)))
        (else false)))

(define (prime-test n)
  (prime-test-iter n (- n 1)))
```

- Results:

```
(prime-test 561)
#t
> (prime-test 1105)
#t
> (prime-test 1729)
#t
> (prime-test 2465)
#t
> (prime-test 2821)
#t
> (prime-test 6601)
#t
```

- They all fool the Fermat test

Exercise 1.28

```
;definitions
(define (even? y)
  (= 0 (remainder y 2)))

(define (square x)
  (* x x))

;expmod and check for nontrivial root
(define (rab-miller-test base exp m)
  (define (exp-mod-check pre-sqr)
    (define (check-4-nontriv pre-sqr sqr)
      (if (and (not (= pre-sqr 1))
               (not (= pre-sqr (- m 1)))
               (= sqr 1))
          0
          sqr))
```

```
     (check-4-nontriv pre-sqr (remainder (square pre-sqr) m)))
  ;modified expmod for check
  (cond ((= exp 0) 1)
        ((even? exp)
            (exp-mod-check (rab-miller-test base (/ exp 2) m)))
        (else (remainder (* base
                            (rab-miller-test base (- exp 1) m))
                          m))))

;check if expmod signaled a non-triv root or if the term equals
;one and completes fermat's theorem
(define (rab-mill-test-recursion n)
  (define (try a)
    (define (check term)
      (and (not (= term 0)) (= term 1)))
    (check (rab-miller-test a (- n 1) n)))
  (try (if (> n 4294967087)
           (+ 1 (random 4294967087))
           (+ 1 (random (- n 1))))))

;iteration of the algorithm to improve probability of certainty
;of prime
(define (prime-iter n times)
  (cond ((< n 2) (display "try a number 1<n<infinity"))
        ((= times 0) true)
        ((rab-mill-test-recursion n) (prime-iter n
                                                 (- times 1)))
        (else false)))

;100 is arbitrary times to perform, but provides a very high
;probability of a prime number being prime
(define (prime? n)
  (prime-iter n 100))
```