SECTION 1

RECURSION

in recursion we just call the function in on itself till a condition is met, this results in the program firstly processing *ALL* the data and then computing one by one

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

;(factorial 6)
;(* 6 (factorial 5))
;(* 6 (* 5 (factorial 4)))
;(* 6 (* 5 (* 4 (factorial 3))))
;(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
;(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))  <-- accumulates all the processes
;(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
;(* 6 (* 5 (* 4 (* 3 2))))
;(* 6 (* 5 (* 4 6)))
;(* 6 (* 5 24))
;(* 6 120)
;720
```

ITERATION

Iteration means going forward as opposed to the recursion's going backwards at first, to the very lowest level, and then going forward calculating the result on the way back up, you *change* the value of the variable at every step:

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

;(fact 4)
;(iter 1  1  4)
;(iter 1 2 4)
;(iter 2 3 4)
;(iter 6 4 4)
;(iter 24 5 4)
;24
```

Product	Input	Counter
1	4	1
1	4	2
2	4	3
6	4	4
24	4	5

DIFF BETWEEN THEM

One of the primary ways to differentiate between an iterative and recursive process is to imagine what'd happen if you turned the computer off, then resumed the current computation.
In a recursive process we can never know how deep into the recursion we are.
In the factorial example, this means that we can't return the right value.
In an iterative process, all state information is contained by the process.

Exercise 1

What is the result printed by the interpreter in response to each expression?

```
10
;10
(+ 5 3 4)
;12
(- 9 1)
;8
(/ 6 2)
;3
(+ (* 2 4) (- 4 6))
;6

(define a 3)
(define b (+ a 1))
(+ a b (* a b))
;(+ 3 (+ 3 1) (* 3 (+ 3 1)))
;(+ 3 4 12)
;(19)

(= a b)
;FALSE (amogus)
(if (and (> b a) (< b (* a b)))
b
a)
;(if (and true true) b a)
;(if true b a)
;(b)
;4

(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))
;(+ 6 7 3)
;16
```

```
(+ 2 (if (> b a) b a))
;(+ 2 (if true b a))
;(+ 2 4)
;(+ 6)

(* (cond ((> a b) a)
     ((< a b) b)
     (else -1))
   (+ a 1))
;(* 4 (+ 3 1))
;(* 4 4)
;16
```

Exercise 2

Translate the following expression into prefix

$$\frac{5+4+(2-(3-(6+\frac{4}{5})))}{3(6-2)(2-7)}.$$

```
(/ (+ 5 4 (- 2 (- 3 (+ 6 (/ 4 5))))) (* 3 (- 6 2) (- 2 7)))
; -37/150
```

Exercise 3

Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers

```
(define (f1 a b c)
  (if (< a b)
      (if (< a c)
          (+ (* b b) (* c c))
          (+ (* b b) (* a a)))
      (if (< b c)
          (+ (* a a) (* c c))
          (+ (* a a) (* b b)))))
```

Exercise 4

Observe that our model of evaluation allows for combinations whose operators are compound expres- sions. Use this observation to describe the behavior of the following procedure:

```
(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

```
;this is just a+|b| by performing a+b if b>0, else a-b
(a-plus-abs-b 3 4)
;(7)
(a-plus-abs-b 3 -4)
```

Exercise 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 defines the following two procedures:

```
(define (p) (p))
(define (test x y) (if (= x 0) 0 y))
```

Then he evaluates the expression `(test 0 (p))` What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer. (Assume that the evaluation rule for the special form if 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.

evaluating (p) will result in an infinite loop for applicative order because internal reduction happens **BEFORE** the values are evaluated, like:

```
(test 0 (p))
(if (= 0 0) 0 (p)) ;<---- here the program will go for an inifnite loop
because it can't evaluate `(p)`
```

for normal order eval this will happen:

```
(test 0 (p))
(if (= 0 0)
    0       ;<--- here the program will stop
    (p))    ;<--- the program will actually never reach this phase
```

Exercise 6

Alyssa P. Hacker doesn't see why if needs to be provided as a special form. "Why can't I just define it as an ordinary procedure in terms of cond?" she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of if:

```
(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-clause)))
```

Eva demonstrates the program for Alyssa:

```
(new-if (= 2 3) 0 5)
5

(new-if (= 1 1) 0 5)
0
```

Delighted, Alyssa uses new-if to rewrite the square-root program:

```
(define (sqrt-iter guess x)
  (new-if (good-enough? guess x)
          guess
          (sqrt-iter (improve guess x) x)))
```

What happens when Alyssa attempts to use this to compute square roots? Explain.

in the old `(if cond A B)` the B condition would only be evaluated if cond was false BUT in the new `(new-if cond A B)` as per the definition the arguement B needs to be evaluated regardless of if the cond was false because when a function is called the first thing that Scheme does with the argument list is evaluating every single arguement

do you remembah? the scheme uses applicative-order for function in septembah?

Exercise 7

The 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 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 the guess. Design a square-root procedure that uses this kind of end test. Does this work better for small and large numbers?

if y is our guess for $\sqrt{x}$ then, $y = \sqrt{x} + \delta y$ and $|y^2 - x| > \varepsilon$, then for small values:

$$\varepsilon = |y^2 - x|$$

$$\varepsilon = |\left(\sqrt{x} + \delta y\right)^2 - x|$$

$$\varepsilon = |(x + 2\delta y)\sqrt{x} - x|$$

$$\varepsilon = 2|\delta y|\sqrt{x}$$

```
(define (average x y) (/ (+ x y) 2))

(define (improve guess x) (average guess (/ x guess)))

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

(define (abs x)
  (cond ((= x 0) 0)
        ((< x 0) (-x)
        ((> x 0) (x)))))

(define (good-enough? guess x)
(< (abs (- (square guess) x)) 0.001))

(define (sqrt-iter guess x)
  (if (good-enough? guess x) guess
      (sqrt-iter (improve guess x) x)))

(define (my-sqrt x)
  (sqrt-iter 1.0 x))

; Given the last and 2nd last guesses, check if the fractional change is good
enough.
; I'm choosing the formula |guessL-guessLL|/guessL < 0.001
(define (good-enough2? guessL guessLL)
  (< (/ (abs (- guessL guessLL)) guessL) 0.01))

(define (sqrt-iter2 guessL guessLL x)
  (if (good-enough2? guessL guessLL)
      guessL
      (sqrt-iter2 (improve guessL x) guessL x)))

(define (my-sqrt2 x)
  (sqrt-iter2 1.0 1.1 x))
```

Exercise 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 procedure analogous to the square-root procedure.

```
(define (improve y x)
  (/ (+ (/ x (* y y)) (* 2 y)) 3))

; Fractional good-enough like in ex 1-7
(define (good-enough? guessL guessLL)
  (< (/ (abs (- guessL guessLL)) guessL) 0.01))

(define (curt-iter guessL guessLL x)
  (if (good-enough? guessL guessLL)
      guessL
      (curt-iter (improve guessL x) guessL x)))

(define (my-curt x)
  (curt-iter 1.0 1.1 x))

; 1.4422495703074083
(my-curt 3)
```

Exercise 9

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

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

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

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

```
(define (+ a b)
  (if (= a 0)
      b
      (inc (+ (dec a) b))))
;recursive process because it doesn't change the value of the variable
;(+ 4 5)
;(inc (+ 3 5))
;(inc (inc (+ 2 5)))
;(inc (inc (inc (+ 1 5))))
;(inc (inc (inc (inc (+ 0 5)))))
;(inc (inc (inc (inc 5))))
;(inc (inc (inc 6)))
;(inc (inc 7))
;(inc 8)
;9

(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))
;iterative because it changes the value of the variable at every step
;(+ 4 5)
;(+ 3 6)
;(+ 2 7)
;(+ 1 8)
;(+ 0 9)
;9
```

Exercise 10

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

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

What are the values of the following expressions?

```
(A 1 10)
(A 2 4)
(A 3 3)
```

Consider the following procedures, where A is the procedure defined above:

```
(define (f n) (A 0 n))
(define (g n) (A 1 n))
(define (h n) (A 2 n))
(define (k n) (* 5 n n))
```

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

```
(A 1 10)
;1024
(A 2 4)
;65536
(A 3 3)
;65536

(define (f n) (A 0 n))
;this is just x 0 always so it is just a function that doubles the arguement
i.e n*2
(define (g n) (A 1 n))
;(A 1 0) is 0
;(A 1 1) is 2
;(A 1 2) is (* 2 2) = 4
;(A 1 3) is (* 2 (* 2 2)) = 8
;(A 1 n) is 2^n
(define (h n) (A 2 n))
;(A 2 0) is 0
;(A 2 1) is 2
;(A 2 2) is 4
;(A 2 3) is 16
;this is (A 1 (A 2 (- n 1)))
;or square of (A 2 (- n 1))
;so this is just square of square of square of square and so on
```

Exercise 11

A function f is defined by the rule that $f(n) = n$ if $n < 3$ and $f(n) = f(n-1) + 2f(n-2) + 3f(n-3)$ if $n \geq 3$. Write a procedure that computes f by means of a recursive process. Write a procedure that computes f by means of an iterative process.

recursive is pretty easy

```
(define (f n)
  (if (< n 3)
      n
      (+ (f (- n 1)) (* (f (- n 2)) 2) (* (f (- n 3)) 3))))
```

for iterative. We need three pieces of state to track the current, the last and the values of f. (that is, $f(n-1)$, $f(n-2)$ and $f(n-3)$.)

```
(define (f n)
  (f-it-helper 2 1 0 n))

(define (iterator a b c counter)
  (if (= counter 0)
      c
      (iterator (+ a (* 2 b) (* 3 c))
                a
                b
                (- counter 1))))
;think like this
;f(0) = 0  |
;f(1) = 1  | all known values
;f(2) = 2  |

;f(3) = f(2) + 2f(1) + 3f(0)
;f(4) = f(3) + 2f(2) + 3f(1)
;f(5) = f(4) + 2f(3) + 3f(2)
;f(6) = f(5) + 2f(4) + 3f(3)
```

Exercise 12

Write a recursive procedure that computes elements of Pascal's triangle.

```
(define (pascal row col)
  (cond ((= col 0) 1)
        ((= col row) 1)
        (else (+ (pascal (- row 1) col) (pascal (- row 1) (- col 1))))))
;idk how to display it :P
```

Exercise 13

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

Hint: Let $\psi = \frac{1\sqrt{5}}{2}$. Use induction and the definition of the Fibonacci numbers to prove that $\text{Fib}(n) = \frac{\varphi^n - \psi^n}{\sqrt{5}}$.

TODO

Exercise 14

Draw the tree illustrating the process generated by the count-change procedure of 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?

```
;the code was this
(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-denomination kinds-of-coins))
                      kinds-of-coins)))))
(define (first-denomination 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)))
(count-change 11)
```

$$\sum_{i=a}^{b} \frac{1}{i(i+2)}$$

Exercise 15

The sine of an angle (specified in radians) can be computed by making use of the approximation $\sin(x) = x$ if x is sufficiently small, and the trigonometric identity to reduce the size of the argument of sin. (For purposes of this exercise an angle is considered "sufficiently small" if its magnitude is not greater than 0.1 radians.) These ideas are incorporated in the following procedures:
we are using the identity $\sin(3x) = 3\sin(x) - 4\sin^3(x)$

```
(define (cube x) (* x x x))
(define (p x)
  (- (* 3 x) (* 4 (cube x))))
(define (sine angle)
  (if (not (> (abs angle) 0.1))
      angle
      (p (sine (/ angle 3.0)))))
```

- How many times is the procedure p applied when (sine 12.15) is evaluated?
- What is the order of growth in space and number of steps (as a function ofa) used by the process generated by the sine procedure when (sine a) is evaluated?

- the procedure p is evaluated 5 times

$$\frac{12.15}{3^n} = 0.1$$

$$n = \frac{\log\left(\frac{12.5}{0.1}\right)}{\log(3)} \approx 4.4$$

```
;also the output is this
(sine 12.5)
-0.060813768577286265
```

- the order of growth of steps is just $\Theta(\log(a))$ and similar in space because there is no recursion

Exercise 16

Design a procedure 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 $b^{\left(\frac{n}{2}\right)^2} = \left(b^2\right)^{\frac{n}{2}}$ 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.)

```
(define (exponent base n)
  (helperiter 1 base n))
(define (helperiter acc base n)
  (cond ((= n 0) acc)
        ((= (remainder n 2) 0) (helperiter acc (* base base) (/ n 2))) ;if the
exponent is a multiple of 2 then do b^n = (b*b)^(n/2)
        (helperiter (* acc base) base (- n 1))))
```

Exercise 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 procedure (in which it is assumed that our language can only add, not multiply) is analogous to the expt procedure:

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

this algorithm takes a number of steps that is linear in b. Now suppose we include, together with addition, operations double, which doubles an integer, and halve, which divides an (even) integer by 2. Using these, design a multiplication procedure analogous to fast-expt that uses a logarithmic number of steps.

we can do something like $ab = 2a\frac{b}{2}$ for even b (because we are given the procedures *double* and *half*)
and the previous same procedure for odd b

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

(define (* a b)
  (cond ((= b 0) 0)
        ((even? b) (* (double a) (half 2)))
        (else (+ a (* a (- b 1))))))
```

Exercise 18

Using the results of Exercise 1.16 and Exercise 1.17, devise a procedure that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps.

we know that in iterative processes we need an accumulator a counter and some number of other arguements for operations, so let's make one of the multiplication arguements as the counter, something like

```
(define (multiply a b)
  (define (iterator acc a b)
    (cond ((= b 0) acc)
          ((even? b) (iterator acc (double a) (half b)))
          (else (iterator (+ acc a) a (- b 1)))))
  (iterator 0 a b))
```

Exercise 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 and b in the fib-iter process of 1.2.2:

$$a \leftarrow a + b$$

$$b \leftarrow a$$

Call this transformation T, and observe that applying T over and over again n times, starting with 1 and 0, produces the pair $\text{Fib}(n+1)$ and $\text{Fib}(n)$. 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 $p = 0$ and $q = 1$ in a family of transformations T_{pq} where T_{pq} transforms the pair (a, b) according to:

$$a \leftarrow bq + aq + ap$$

$$b \leftarrow bp + aq$$

Show that if we apply such a transformation T_{pq} twice, the effect is the same as using a single transformation $T_{p'q'}$ of the same form, and compute p' and q' in terms of p and q. This gives us an explicit way to square these transformations, and thus we can compute T^n using successive squaring.

```
(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
               ⟨??⟩ ; compute p′
               ⟨??⟩ ; compute q′
               (/ count 2)))
    (else (fib-iter (+ (* b q) (* a q) (* a p))
                    (+ (* b p) (* a q))
                    p
                    q
                    (- count 1)))))
```

let us say there is a matrix T_{pq}, then according to the question this matrix *transforms* the matrix (in question called pair) $\begin{pmatrix} a \\ b \end{pmatrix}$ as follows:

$$T_{pq} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} bq + aq + ap \\ bp + aq \end{bmatrix}$$

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

$$T_{pq} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \begin{bmatrix} p+q & q \\ q & p \end{bmatrix}$$

$$T_{pq} = \begin{bmatrix} p+q & q \\ q & p \end{bmatrix}$$

and according to question we need to show that T_{pq}^2 (T_{pq} used 2 times) is equivalent to some $T_{p'q'}$, so:

$$T_{pq}^2 = \begin{bmatrix} p+q & q \\ q & p \end{bmatrix}^2$$

$$T_{pq}^2 = \begin{bmatrix} p^2 + 2pq + 2q^2 & 2pq + q^2 \\ 2pq + q^2 & q^2 + p^2 \end{bmatrix}$$

$$T_{pq}^2 = T_{(2pq+q^2)(q^2+p^2)}$$

now just shove it in the code and *noice*

```
(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
               (+ (* 2 p q) (* q q)) ; compute p'
               (+ (* q q) (* p p)) ; compute q'
               (/ count 2)))
    (else (fib-iter (+ (* b q) (* a q) (* a p))
                    (+ (* b p) (* a q))
                    p
                    q
                    (- count 1)))))
```

Exercise 20

> The process that a procedure generates is of course dependent on the rules used by the interpreter. As an example, consider the iterative gcd procedure given above. Suppose we were to interpret this procedure using normal-order evaluation, as discussed in 1.1.5. (The normal-order-evaluation rule for if 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?

- for normal order evaluation it will be

```
;i AM NOT WRITING THAT WTF it will take ages bro
```

- for applicative order of evaluation

```
; ok fuck this i saw everywhere and everyone has a different answer? wtf?
```

Exercise 21

> Use the smallest-divisor procedure to find the smallest divisor of each of the following numbers: 199, 1999, 19999.

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

(define (divides? a b)
  (= (remainder b a) 0))

(define (find-divisor n test-divisor)
  (cond ((> (* test-divisor test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(smallest-divisor 199)
;199
(smallest-divisor 1999)
;;1999
(smallest-divisor 7)
;7
```

Exercise 22

Most Lisp implementations include a primitive called runtime that returns an integer that specifies the amount of time the system has been running (measured, for example, in microseconds). The following timed-prime-test procedure, when called with an integer prints and checks to see if is prime. If is prime, the procedure prints three asterisks followed by the amount of time used in performing the test.

```
(define (timed-prime-test n)
  (newline)
  (display n)
  (start-prime-test n (runtime)))

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

Using this procedure, write a procedure search-for-primes that checks the primality of consecutive odd integers in a specified range. Use your procedure 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 $\Theta(\sqrt{n})$ 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?

there actually isn't a runtime primitive in racket *(because i couldn't get scheme to work on my machine)*, BUT there is *current-milliseconds* which i *think* does the same stuff?

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

(define (find-divisor n test-divisor)
  (cond ((> (* test-divisor test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(define (divides? a b)
  (= (remainder b a) 0))

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

```
(define (timed-prime-test n start-time)
  (cond ((prime? n)

         (display n)
         (display "is a prime | found in ")
         (display (- (current-milliseconds) start-time))
         (newline)
         true)
        (else false)))

(define (primes-larger-than n x)
  (cond ((= x 0) false)
        (else (cond ((timed-prime-test n (current-milliseconds)) (primes-
larger-than (+ n 1) (- x 1)))
                    (else (primes-larger-than (+ n 1) x))))))

(primes-larger-than 100000000 3)
;100000007 is a prime | found in 0
;100000037 is a prime | found in 0
;100000039 is a prime | found in 0
;#f

(primes-larger-than 1000000000 3)
;1000000007 is a prime | found in 0
;1000000009 is a prime | found in 0
;1000000021 is a prime | found in 0
;#f

(primes-larger-than 10000000000 3)
;10000000019 is a prime | found in 0
;10000000033 is a prime | found in 0
;10000000061 is a prime | found in 0
;#f
```

ummmmm so this is awkward, uhhhhhhh my machine is *too powerful* for the current-milliseconds to be counted so it just returns 0 for everytime welp……fuck

Exercise 23

The `smallest-divisor` procedure 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, define a procedure next that returns 3 if its input is equal to 2 and otherwise returns its input plus 2. Modify the `smallest-divisor` procedure 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?

```
#lang racket

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

(define (find-divisor n test-divisor)
  (cond ((> (* test-divisor test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (nextdiv test-divisor)))))

(define (divides? a b)
  (= (remainder b a) 0))

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

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

(define (timed-prime-test n start-time)
  (cond ((prime? n)
         (display n)
         (display "is a prime | found in ")
         (display (- (current-milliseconds) start-time))
         (newline)
         true)
        (else false)))

(define (primes-larger-than n x)
  (cond ((= x 0) false)
        (else (cond ((timed-prime-test n (current-milliseconds))
                     (primes-larger-than (+ n 1) (- x 1)))
                    (else (primes-larger-than (+ n 1) x))))))
```

```
(primes-larger-than 100000000 3)
;100000007 is a prime | found in 0
;100000037 is a prime | found in 0
;100000039 is a prime | found in 0
;#f

(primes-larger-than 1000000000 3)
;1000000007 is a prime | found in 0
;1000000009 is a prime | found in 0
;1000000021 is a prime | found in 0
;#f

(primes-larger-than 10000000000 3)
;10000000019 is a prime | found in 0
;10000000033 is a prime | found in 0
;10000000061 is a prime | found in 0
;#f
```

uhhhhhh yeah same thing :P

Exercise 24

Modify the timed-prime-test procedure of Exercise 1.22 to use fast-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?

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

(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-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))

(define (fast-prime? n times)
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
```

```
        (else false)))

(define (timed-prime-test n start-time)
  (cond ((fast-prime? n 12)
         (display n)
         (display "is a prime | found in ")
         (display (- (current-milliseconds) start-time))
         (newline)
         #t)
        (else #f)))

(define (primes-larger-than n x)
  (cond ((= x 0) false)
        (else (cond ((timed-prime-test n (current-milliseconds))
                     (primes-larger-than (+ n 1) (- x 1)))
                    (else (primes-larger-than (+ n 1) x))))))

(primes-larger-than 100000 3)
(display "\n\n")
(primes-larger-than 1000 3)

;1000003is a prime | found in 180
;1000033is a prime | found in 186
;1000037is a prime | found in 186
;#f
;
;
;1009is a prime | found in 0
;1013is a prime | found in 0
;1019is a prime | found in 0
;#f
```

hory shit, racket is so slow that it actually gave me a time

Exercise 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

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

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

no clue about time complexity but this sht will not fly for space for sure, cuz fast-expt makes recursive calls to itself and so does remainder, which means the remainder will be called for EACH call to fast-expt (atleast the way we have defined it till now)

Exercise 26

Louis Reasoner is having great difficulty doing Exercise 1.24. His `fast-prime?` test seems to run more slowly than his `prime?` test. Louis calls his friend Eva Lu Ator over to help. When they examine Louis's code, they find that he has rewritten the expmod procedure to use an explicit multiplication, rather than calling square:

```
(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))))
```

"I don't see what difference that could make," says Louis. "I do." says Eva. "By writing the procedure like that, you have transformed the $\Theta(\log n)$ process into a $\Theta(n)$ process." Explain.

if we use `(square (expmod base (/ exp 2) m))` then firstly we would've done then (expmod base (/ exp 2) m) and then (square <result> <result>) so we would've evaluated (expmod) ONCE every cycle

but if we do `(* (expmod base (/ exp 2) m) (expmod base (/ exp 2) m))` then we need to calculate (expmod) TWICE which means that for every

Exercise 27

Demonstrate that the Carmichael numbers listed in Footnote 47 really do fool the Fermat test. That is, write a procedure that takes an integer n and tests whether a^n is congruent to a modulo n for every $a < n$, and try your procedure on the given Carmichael numbers.

Numbers that fool the Fermat test are called Carmichael numbers, and li̇̈le is known about them other than that they are extremely rare. ǥere are 255 Carmichael numbers below 100,000,000. ǥe smallest few are 561, 1105, 1729, 2465, 2821, and 6601. In testing primality of very large numbers chosen at random, the chance of stumbling upon a value that fools the Fermat test is less than the chance that cosmic radiation will cause the computer to make an error in carrying out a "correct" algorithm. Considering an algorithm to be inadequate for the first reason but not for the second illustrates the difference between mathematics and engineering.

Exercise 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 integer less than n, then $a^{n-1} \equiv 1(\mathrm{mod})n$. To test the primality of a number n by the Miller-Rabin test, we pick a random number $altn$ and find $a^{n-1}(\mathrm{mod})n$ using the `expmod` procedure. 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 $n-1$ whose square is equal to 1 modulo n. It is possible to prove that if such a nontrivial square root of 1 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 procedure to signal if it discovers a nontrivial square root of 1, and use this to implement the Miller-Rabin test with a procedure analogous to `fermat-test`. Check your procedure by testing various known primes and non-primes. Hint: One convenient way to make `expmod` signal is to have it return 0.

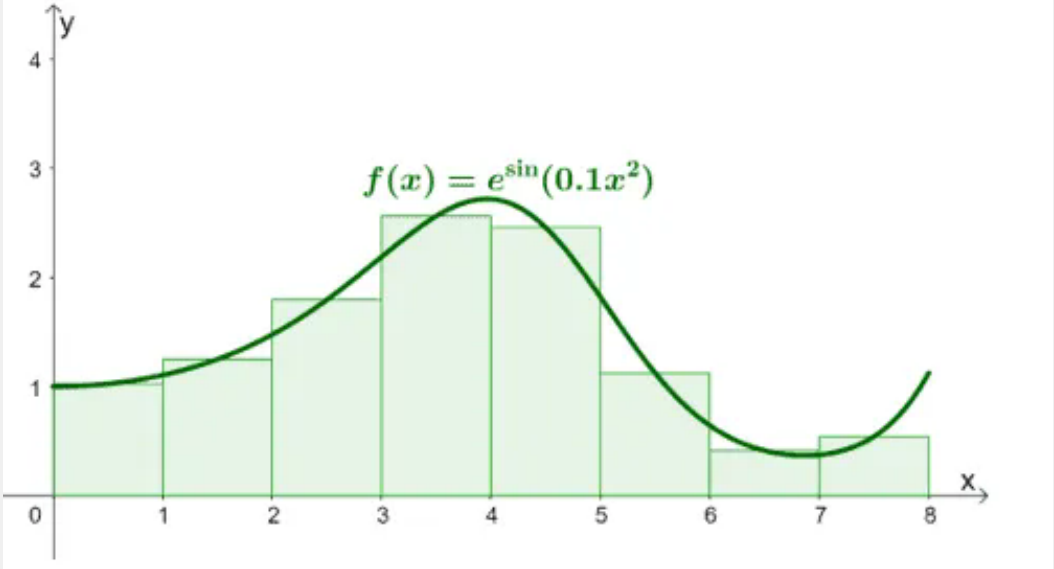
TODO

```
(define (sum term a next b)
  (if (> a b) 0
      (+ (term a) (sum term (next a) next b))))
```

$$\sum_{k=a}^{b} f(n) = f(a) + ... + f(b)$$

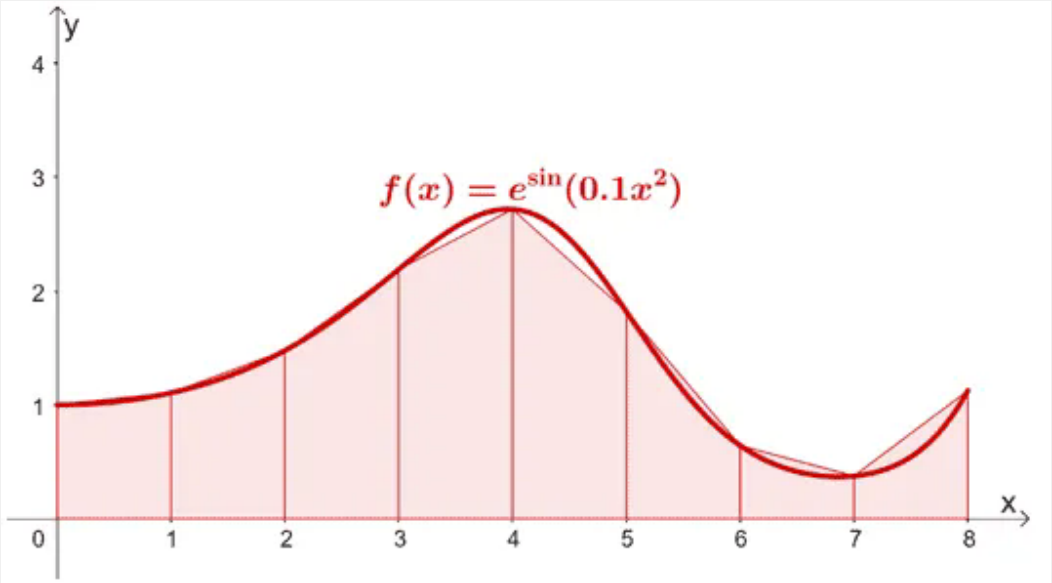
MIDPOINT THEOREM

$$\int_a^b f(x) = dx\left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right)\right]$$



TRAPEZOIDAL THEOREM

$$\int_z^b f(x) = \frac{dx}{2}[f(a) + 2(f(a + dx) + f(a + 2dx) + f(a + 3dx) + f(a + 4dx)...)f(b)]$$



Exercise 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 and b is approximated as:

$$\left(\frac{h}{3}\right)(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n)$$

where $h = \frac{b-a}{n}$, for some even integer n, and $y_k = f(a + kh)$. (Increasing n increases the accuracy of the approximation.) Define a procedure that takes as arguments f , a, b, and n and returns the value of the integral, computed using Simpson's Rule. Use your procedure to integrate cube between 0 and 1 (with $n = 100$ and $n = 1000$), and compare the results to those of the integral procedure shown above.

```
(define (integral f a b n)

  (define (sum term a next b)
    (if (> a b)
        0
        (+ (term a)
           (sum term (next a) next b))))

  (define (h) (/ (- b a) n))

  (define (y k)
    (f (+ a (* k (h)))))

  (define (element k)
    (cond ((= k 0) (* 1 (y k)))
          ((= k n) (* 1 (y k)))
          ((odd? k) (* 4 (y k)))
          (else (* 2 (y k)))))

  (define (sus n)
    (+ n 1))

  (* (/ (h) 3) (sum element 0 sus n)))

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

(integral square 0 1 100.00)
;0.33333333333333337
(integral square 0 1 1000.00)
;0.33333333333333326
```

HOLY SHIT THIS TOOK ME A WHOLE DAY TO DO, i have no idea about calc 1 i guess ;-; also just discovered that you need double decimal digits after the number to get accurate decimal places :P }:3

Exercise 30

ǫe sum procedure above generates a linear recursion. ǫe procedure can be rewriǟen so that the sum is performed iteratively. Show how to do this by filling in the missing expressions in the following definition:

```
(define (sum term a next b)
  (define (iter a result)
    (if (??)
        (??)
        (iter (??)(??))))
  (iter (??)(??)))
```

```
(define (sum term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (+ result (term a)))))
  (iter a 0))
```

Exercise 31

- ǫe sum procedure is only the simplest of a vast number of similar abstractions that can be captured as higher-order procedures. Write an analogous procedure 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 π using the formula.

$$\frac{\pi}{4} = \frac{2.4.4.6.6.8...}{3.3.5.5.7.7...}$$

- If your product procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

- ```
  (define (products func a next b)
    (if (> a b) 1
        (* (func a) (products func (next a) next b))))

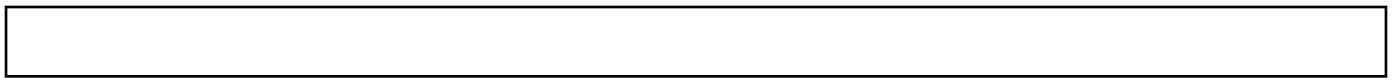
  (define (factorial n)
    (define (next-term n)
  ```
  ```

```
      (+ n 1))
    (define (element k)
      k)
    (products element 1 next-term n))

  (factorial 3)
```

- ```
  (define (products-iter func a next b)
    (define (helperiter a result)
      (if (> a b) result
          (helperiter (next a) (* result (func a)))))
    (helperiter a 1))
  ```

Exercise 32


  ```