1. (**10 points.**) Define SCHEME functions with the following specifications.

   (a) (**2 points.**) Define a SCHEME function `max2` which takes two numeric inputs (call them x and y) and returns the larger of the two. (You may not use the built-in scheme function `max` for this purpose—define your own function from scratch using a conditional.)

```
(define (max2 x y)
  (if (> x y) x y))
```

   (b) (**2 points.**) Define a SCHEME function `max3` which takes three numeric inputs (call them x, y, and z) and returns the largest of the three. (You may not use the built-in scheme function `max` for this purpose, but you may use your function `max2` from the previous problem.)

```
(define (max3 x y z)
  (max2 (max2 x y) z))
```

   (c) (**2 points.**) Define a SCHEME function `crazy` which takes a single input x and returns

$$\frac{(x+10)(x+10)+10}{x+10}.$$

```
(define (crazy x)
  (define (p10 y) (+ y 10))
  (/ (p10 (* (p10 x) (p10 x)))
     (p10 x)))
```

   (d) (**2 points.**) Define a SCHEME function `monster-fact` which takes a single numeric argument x and returns $(x!)!$. (Yes, that's the factorial of the factorial of x, so `(monster-fact 4)` should return $(4!)! = 24! = 620448401733239439360000$. You must define the `factorial` function from scratch, if you intend to use it.)

```
(define (monster-fact k)
  (define (factorial n)
    (if (= n 0)
        1
        (* n (factorial (- n 1)))))
  (factorial (factorial k)))
```

   (e) (**2 points.**) Define a SCHEME function `dfact` (which stands for "double factorial"). The double factorial function is defined (for the natural numbers $\{0, 1, 2, \ldots\}$) by the recursive rule:

$$\mathrm{dfact}(n) = \begin{cases} 1 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ n \cdot \mathrm{dfact}(n-2) & \text{otherwise.} \end{cases}$$

```
(define (dfact n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (else (* n (dfact (- n 1))))))
```

2. (**10 points.**) In this problem, you will implement *Halley's method* for extracting square roots. Consider the following mysterious function of two variables Q and g:

$$h(Q,g) = \frac{g(g^2+3Q)}{3g^2+Q}.$$

Edmond Halley (after whom the comet is named) noticed the following remarkable fact. For any (positive) number Q and any (positive) number g, $h(Q,g)$ is always closer to $\sqrt{Q}$ than g was. In particular, for any specific fixed value of Q we can consider the sequence of numbers

$$g_s = \begin{cases} 1 & \text{if } s = 0, \\ h(Q, g_{s-1}) & \text{for } s > 0. \end{cases}$$

Then these numbers $g_0, g_1, \ldots$ converge very quickly to $\sqrt{Q}$.

(a) (**1 points.**) Write a SCHEME function `h` which computes the function above (so, the function `h` should take two arguments Q and g).

```
(define (h Q g)
  (/ (+ (* g g g) (* 3 g Q))
     (+ (* 3 g g) Q)))
```

(b) (**3 points.**) Write a function `Halley-iterate` which takes two numeric arguments, Q and t, and returns g_t (defined by the sequence above). Thus, if $t = 0$, your function should return 1; if $t = 1$, your function should return $g_1 = h(Q,1)$, etc.

```
(define (Halley-iterate Q t)
  (cond ((= t 0) 1)
        (else (h Q (Halley-iterate Q (- t 1))))))
```

(c) (**4 points.**) Write a function `Halley-approx` which takes a single numeric argument Q and returns an approximation α to the square root of Q with the property that $|\alpha^2 - Q| \leq .00001$. To do this, your function should effectively compute the sequence $g_0, g_1, g_2, \ldots$ until it finds a g_t for which $|g_t^2 - Q| \leq .00001$ (at which point it can simply return the value g_t).

```
(define (Halley-approx Q)
  (define (Halley-work guess)
    (if (< (abs (- Q (* guess guess))) .00001)
        guess
        (Halley-work (h Q guess))))
  (Halley-work 1))
```

(d) (**2 points.**) Show how to restructure your code from the previous problems so that h is defined in the scope of `Halley-approx`, and use this restructuring to make h a function of a single parameter g.

```
(define (Halley-approx Q)
  (define (h g)
    (/ (+ (* g g g) (* 3 g Q))
```

```
        (+ (* 3 g g) Q)))
  (define (Halley-work guess)
    (if (< (abs (- Q (* guess guess))) .00001)
        guess
        (Halley-work (h guess))))
  (Halley-work 1))
```

3. (**10 points.**) The integers 1, 2, 4, and 5 can be written as the *sum of two perfect squares*:

$$1 = 0^2 + 1^2, \quad 2 = 1^2 + 1^2, \quad 4 = 0^2 + 2^2, \quad \text{and} \quad 5 = 1^2 + 2^2.$$

On the other hand, neither 3, 6, nor 7 can be expressed this way.

In this problem, you will define a function `sum-of-squares` so that `(sum-of-squares n)` returns `#t` if the positive integer n can be written as a sum of squares of two integers and `#f` otherwise. You may assume that n is positive. If you wish, you may use the following function `is-square`, which returns `#t` if k is a perfect square, and `#f` otherwise:

```
(define (is-square k) (let ((int-root (round (sqrt k))))
                        (= k (* int-root int-root))))
```

(a) (**2 points.**) Define a SCHEME function `square-pieces` so that `(square-pieces x n)` returns `#t` if both x and $(n - x)$ are perfect squares, and `#f` otherwise.

```
(define (square-pieces x n)
        (and (is-square x)
             (is-square (- n x))))
```

(b) (**4 points.**) Observe that n can be written as a sum of two squares exactly when there is a number $x \in \{0, \ldots, n\}$ for which `(square-pieces x n)` returns `#t`. Write a SCHEME function `(test-upto k n)` which returns `#t` if there is a number $x \in \{0, \ldots, k\}$ for which `(square-pieces x n)` is true.

```
(define (test-upto k n)
  (if (< k 0) #f
             (or (square-pieces k)
                 (test-upto (- k 1) n))))
```

(c) (**4 points.**) Using the above functions, define the SCHEME function `sum-of-squares`. For full credit, indicate how the definitions of your helper functions can be made private; do you need to keep passing around the parameter `n`?

```
(define (sum-of-squares n)
  (define (is-square k) (let ((int-root (round (sqrt k))))
                          (= k (* int-root int-root))))
  (define (square-pieces x)
          (and (is-square x)
               (is-square (- n x))))
  (define (test-upto k)
    (if (< k 0) #f
               (or (square-pieces k)
                   (test-upto (- k 1)))))
  (test-upto (ceiling (sqrt n))))
```

4. (**10 points.**)

(a) (**2 points.**) Consider the following version of `factorial`:

```
(define (factorial n)
  (let ((recursive-value (factorial (- n 1))))
    (if (= n 0)
        1
        (* n recursive-value))))
```

For which, if any, values of n does this correctly compute $n!$? Explain.

None. The functional *always* generates a call to `(factorial (- n 1))` and hence generates an infinite sequence of calls regardless of n.

(b) (**2 points.**) Consider the following declaration:

```
(define (f x)
  (define (g y) (+ x y))
  (define (h x) (+ x (g x)))
  (h (+ x 10)))
```

After this, what would `(f 100)` return?

It returns 320.

(c) (**2 points.**) To what does the following expression evaluate?

```
(let ((x 10)
      (y 20)
      (z 40))
  (let ((x (+ x 10))
        (y (+ x 20)))
    (+ z (- x y))))
```

It returns 30.

(d) (**2 points.**) Consider the following two implementations of the familiar function *times*.

```
(define (times x y)
  (cond ((= x 0) 0)
        ((= y 0) 0)
        (else (+ x (times x (- y 1))))))
```

and

```
(define (double x) (* 2 x))
(define (half x) (floor (/ x 2)))
(define (even? x) (= 0 (modulo x 2)))
(define (ftimes x y)
  (cond ((= x 0) 0)
        ((even? x) (double (ftimes (half x) y)))
        (else      (+ y (ftimes (- x 1) y)))))
```

The both correctly compute multiplication (if the arguments are non-negative integers). However, the behavior of a call to `(times 1000000 1000001)` will be quite different from that of a call to `(ftimes 1000000 1000001)`. Explain.

The first one (`times`) will carry out y actual multiplications. If y is large, this can be slow. The second one `ftimes` reduces y by a factor one-half after no more than two calls, and hence generates a number of calls that grows with the number of bits of y, rather than y itself. This code can multiply large numbers efficiently.

(e) (**2 points.**) Consider the following flawed implementation of the Fibonacci sequence.

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

For which values, if any, of n will this correctly compute the nth Fibonacci number? What's wrong?

It works for 0, but no other inputs—the problem is that there is no n for which both recursive calls (to $n-1$ and $n-2$) are both determined by the code.

Scratch Space

Scratch Space