RECURSION 0.3

COMPUTER SCIENCE 61AS

The Basics of Recursion

1. What are three things you find in every recursive function?

Solution:

(a) One or more base cases

(b) Way(s) to make the problem into a smaller problem of the same type (so that it can be solved recursively).

(c) One or more recursive cases that solve the smaller problem and then uses that to solve the original (large) problem.

2. (a) What does the mathematical function factorial do? Write out your answer in either math or words.

Solution:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 2 \cdot 1$$

(b) Below is a Racket function that computes factorial. What is its domain and range? Identify the three things from question 1.

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

Solution: The domain is integers, and the range is also integers.

1. Base case: `(<= n 1)`
2. Making the problem smaller: `(- n 1)`
3. Recursive case: `(* n (factorial (- n 1)))`

(c) Write out the recursive calls made when we do `(factorial 4)`.

Solution:

```
(factorial 4) => (* 4 (factorial 3))
(factorial 3) => (* 3 (factorial 2))
(factorial 2) => (* 2 (factorial 1))
(factorial 1) => 1
(factorial 2) => (* 2 (factorial 1)) => (* 2 1) => 2
(factorial 3) => (* 3 (factorial 2)) => (* 3 2) => 6
(factorial 4) => (* 4 (factorial 3)) => (* 4 6) => 24
```

3. Below is a Racket function that computes the n^{th} Fibonnaci number.

(a) What is its domain and range? Also, identify the three things from question 1.

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

Solution: The domain is integers and the range is also integers.

1. Base cases: `(= n 0)` AND `(= n 1)`
2. Making the problem smaller: `(- n 1)` AND `(- n 2)`
3. Recursive case: `(+ (fib (- n 1)) (fib (- n 2)))`. Note, this is a single recursive case with TWO recursive calls.

(b) Write out the recursive calls made when we do `(fib 4)`. (This will look like an upside-down tree).

Solution:

```
              (fib 4)
               /    \
```

```
          (fib 3)     (fib 2)
         /     |       |     \
   (fib 2) (fib 1) (fib 1) (fib 0)
    /    \
(fib 1) (fib 0)
```

Practice with Recursion

1. What's wrong with the following code? Identify all of the things and fix them.

```
;; Multiplies two numbers together without using *
(define (multiply x y)
    (if (= x 0)
        1
        (+ y (multiply x y))))
```

Solution: There are two errors.

First, the base case should return 0 instead of 1. This can be most easily verified by calling the `multiply` function with first argument equal to 0. The old code would have returned 1 instead of the correct answer, 0.

Second, the recursive call should be `(multiply (- x 1) y)`. The current code does NOT make the problem smaller, which would lead to an infinite loop.

2. We're going to write a function `square-sent` that takes in a sentence of numbers and squares each element.

   (a) What should our base case be? (Hint: when dealing with a numerical input, our base case is usually when the input is a 0 or 1. What might the correspondance be for when a sentence is the input?)

Solution: When the sentence is empty, we should return the empty sentence. In code, this would be written:

```
(if (empty? sent)
    '()
    ...)
```

(b) Write the function `square`.

Solution:

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

(c) If the first number in our sentence is a 3, what do we want the first number in our output to be?

Solution: 9, which is simply 3 squared.

(d) Let's generalize this a little more. If our sentence is the variable `sent`, what do we want the first number in our output to be?

Solution: `(square (first sent))`

(e) Assuming we now know how to figure out the first number in our output, what should the rest of the numbers be? (Remember: we have this really nifty function `square-sent` that computes the squares of a sentence of numbers...)

Solution: `(square-sent (bf sent))`

(f) Using the parts you've come up with so far, write the entire `square-sent` function.

Solution:

```
(define (square-sent sent)
    (if (empty? sent)
        '()
        (se (square (first sent))
            (square-sent (bf sent)))))
```

3. Write a procedure `merge`, which accepts two sorted sentences as input, and merges them together to produce a sorted sentence as output. (If you're having trouble with this problem, try breaking it down into steps like the question above!)

```
-> (merge '(2 2 8 19) '(1 5 6 7 10))
(1 2 2 5 6 7 8 10 19)
```

Solution:

```
(define (merge sent1 sent2)
    (cond ((empty? sent1) sent2)
          ((empty? sent2) sent1)
          ((< (first sent1) (first sent2))
           (se (first sent1)
               (merge (bf sent1) sent2)))
          (else (se (first sent2)
                    (merge sent1 (bf sent2))))))
```

4. Write `foobar`. This procedure outputs a sentence of `foo`'s and `bar`'s, in the sequence `foo, bar, bar, foo, bar, bar, foo, ...`. The formal parameter indicates up to which term to output. (Remember, computer scientists start counting from 0!) The `remainder` function may be useful.

```
-> (foobar 4)
(foo bar bar foo bar)

-> (foobar 2)
(foo bar bar)
```

Solution:

```
(define (foobar n)
    (cond ((= n 0) '(foo))
          ((= (remainder n 3) 0)
           (se (foobar (- n 1)) 'foo))
          (else (se (foobar (- n 1)) 'bar))))
```