RECURSION, ITERATION, AND EFFICIENCY 3

COMPUTER SCIENCE 61AS

Basics of Recursion and Iteration

1. What is a tail-recursive call?

   Solution: A tail recursive call is a call to a procedure that does some calculation in the middle of recursing to keep track of intermediate values and to avoid remembering large expressions. An iterative process utilizes tail-recursion and will only need to keep track of a single value that it can update until it reaches a base case. In a recursive process, each recursive call will expand the final expression that needs to be evaluated for its output.

2. Why do we say certain procedures generate recursive and iterative processes when recursion is used either way?

   Solution: Procedures in Racket that use recursion ar calleed recursive procedures. However, there is a difference in the processes that are created by procedures. Dont be confused by the phrases recursion, recursive process and iterative process. You should know what recursion is by now. Both recursive processes and iterative processes are created by recursion - whether it is recursive or iterative depends on whether or not we utilize tail-recursion.

3. What is the primary reason to use iteration instead of recursion?

   Solution: Iteration typically uses less memory than recursion does.

4. What is a disadvantage of using iteration?

   Solution: Iterative procedures is usually harder to read/write.

5. Identify whether each procedure will generate a recursive or an iterative process.

a.
```
(define (fac n)
  (if (= n 1)
      1
      (* n (fac (- n 1)))))
```

Solution: Recursive

b.
```
(define (foo a b)
  (if (< a 0)
      b
      (foo (- a 1) (+ b 1))))
```

Solution: Iterative

c.
```
(define (bar n)
  (define (loop i)
    (if (= i 0)
        nil
        (se (loop (quotient i 2)) (remainder i 2))))
  (loop n))
```

Solution: Recursive (Just because there's a helper procedure does not mean a procedure is iterative!)

Practice with Recursion/Iteration

1. Determine whether each procedure generates a recursive/iterative process. If the procedure generates a recursive process, rewrite it so it generates an iterative one and vice versa for iterative processes.

   a.
```
(define (count-letters sent)
  (count-helper sent 0))
(define (count-helper sent letter-count)
  (if (empty? sent)
      letter-count
      (count-helper (bf sent)
                    (+ letter-count
                       (count (first sent))))))
```

   Solution: Iterative;

```
(define (count-letters sent)
  (if (empty? sent)
      0
      (+ (count (first sent)) (count-letters (bf sent)))))
```

   b.
```
(define (remove-letter letter wd)
  (cond ((empty? wd) "")
        ((eq? (first wd) letter) (remove-letter letter (bf wd)))
        (else (word (first wd) (remove-letter letter (bf wd))))))
```

   Solution: Recursive;

```
(define (remove-letter letter wd)
  (cond ((empty? wd) (bf letter))
        ((eq? (first wd) (first letter))
         (remove-letter letter (bf wd)))
        (else
         (remove-letter (word letter (first wd))
                        (bf wd)))))
```

Basics of Orders of Growth

1. Rank the orders of growth from slowest to fastest: $\theta(n), \theta(1), \theta(n^2), \theta(\log n)$

Solution: $\theta(1), \theta(\log n), \theta(n), \theta(n^2)$

2. If we know that a procedure that has an input of size n runs in $\theta(n^2)$ time, is it possible to determine how long it will take to finish on a given input? Why/Why not?

Solution: No; It is important to remember that orders of growth are not necessarily a measure of how fast the procedure is overall, although you will hear people use it that way (because orders of growth are a measure of how good the procedure is). All it specifies is what will happen for changes in the input size.

3. Will a procedure that runs in $\theta(n^2)$ always run slower than a procedure that runs in $\theta(n)$? Why/Why not?

Solution: No; orders of growth are a way for us to determine how well our procedures scale to bigger inputs. For example, if the order of growth of time for a procedure is $\theta(n^2)$, then we know that if we double the size of the input n, the time taken by the procedure will (approximately) quadruple. If we triple the size of the input, the time taken will increase by a factor of 9.

Suppose we had a $\theta(n)$ procedure that finished in 10 seconds for a given input but we had a $\theta(n^2)$ procedure that finished in 1 second for the same input. If we doubled the input size, the $\theta(n)$ procedure would take 20 seconds and the $\theta(n^2)$ procedure would take 4 seconds, still faster than the $\theta(n)$ procedure. For large inputs, however, we expect that the $\theta(n)$ procedure run faster.

Practice with Orders of Growth

1. Using big-theta notation, classify the order of growth for both time and memory used by each of the procedures below.

a.
```
(define (min-sent sent)
  (if (empty? (bf sent))
      (first sent)
      (min (first sent) (min-sent (bf sent)))))
```

Solution: Time: $\theta(n)$, Memory: $\theta(n)$

b.
```
(define (foo n)
  (define (loop num)
    (if (= num 0)
        ()
        (se (loop (quotient num 2)) (remainder num 2))))
  (loop n))
```

Solution: Time: $\theta(\log n)$, Memory: $\theta(\log n)$

c. A procedure called `all-pair-sums`, which takes a sentence of numbers and returns a sentence of all the sums of every possible pair of numbers in the sentence. Only give the order of growth for time.

```
(all-pair-sums '(1 2 3)) => (3 4 5) ;; (1+2 1+3 2+3)
(all-pair-sums '(2 4 5 6)) => (6 7 8 9 10 11)
```

Solution: Time: $\theta(n^2)$