Week 3 – 04-02-05
Big-O! Explanation of Big Oh, Omega, & Theta

Theta(1) - Constant Time

All this means is that f does the same thing to every input it gets. If f gets one input , or f gets a list of 6000000 inputs, it will still return in (roughly) the same amount of time.

Examples:

```
(define (contant1 sent)
  (first sent))
```

So lets look at constant1. No matter what this function returns the first element in sent. Obviously constant time.

```
(define (constant2 sent)
  (se (first sent) (square (first (bf sent)))))
```

Constant2 does a whole bunch of things, but does not care how many inputs it receives. Whether given a sentence of 2 numbers or 1000 numbers constant2 6 things total. This is also constant.

```
(define (constant3 sent)
  (if (empty? sent)
      '()
      (se (first sent) (bf sent))))
```

Constant3 is a bit trickier. If there are 0 inputs, the procedure does one thing. Otherwise the procedure does 3 things. If I feed 2000000 or 5 arguments to constant3 the procedure will still do only three things (se, first and bf). Constant!

Theta(n) - Linear Time

Linear Time means that f does approximately the same thing to every one of the n arguments it receives. Another way to put it is that f does something to every argument in its arg-list, whether it is checking it or incrementing a counter. Lets look at some examples.

```
(define (factorial n)
      (if (equal? n 1)
           1
           (* n (factorial (- n 1)))))
```

factorial multiplies `n * n-1 * n-2 * .... * 2 * 1`. This is acheived by calling `(fact (- n 1))` as many times as there are "n" 's. The size of the input matters!

```
(define count (lambda (sent)
    (if (empty? sent)
          0
         (+ 1 (count (bf sent))))))).
```

This should obviously be linear.

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

This is trickier... but as we can see linear goes through each item in sent and does the same thing to it. Thus depending on the number of inputs(n), linear does a constant function N-TIMES!

```
(define (linear2 sent)
  (define (baz sent new-list)
    (if (empty? (bf sent))
        new-list
        (baz (bf sent) (se (first sent) new-sent))))
  (baz sent '()))
```

linear2 first calls the internal function baz. Baz is obviously a linear function (it runs through the list and puts each item into the new-list). Thus the total number of calls is `n+1`, which is approximately equal to `n`.

Theta(n^2) - Polynomial Time

This means that for every call to an argument, that argument does something `n` times. i.e. An algorithm where you compare each element to every other will run in $\Theta(n^2)$ time. Here is the classic example: `sort`.

```
(define (sort unsorted)
  (sort-helper unsorted '()))

(define (sorted-helper unsorted sorted-list)
  (if (empty? unsorted)
       sorted-list
      (sorted-helper (bf unsorted)
                     (place-in (first unsorted)
                               sorted-list))))
```

```
(define (place-in num sent)
  (cond ((empty? sent) num)
        ((> (first sent) num) (se num sent))
        (else (se (first sent) (place-in num (bf sent))))))
```

Let's analyze what our sorting algorithm does. sort-helper takes each item one at a time and calls place-in on the item. So far our algorithm runs in n time. place-in then takes the number and places it in its correct place in the new list. To do so, it keeps comparing num to each item in the list, until it finds a number bigger than num. At worst case, num is bigger than every other number. Thus each call to place-in takes n time. Thus we have an algorithm that for each of n items, calls something else n times. Our total time is (roughly) `n + n + n + ... + n`. (`n-times`). We can factor out an `n` and get `n(n)` or n^2 time.

Theta(2^n) - Exponential time

This is the easiest case to analyze. Pretty much, exponential functions make more than one recursive call in each iteration. The trace of the function looks like a tree. i.e. each call to f produces 2 more calls to f.

Here are 2 examples:

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

```
(define (baz n)
  (if (<= n 0)
       1
      (+ (baz (- n 10)) (baz (- n 11))))))
```

Thus, it should be obvious that each call to the above functions produce more than one recursive call to itself. Thus these functions are exponential.

>Why is (last sent) O(n) when (first sent) is O(1)? (just something to think about…)