Order of Growth

Last week we saw some examples of determining the order of growth of functions. For each of the following functions, determine the O, Ω, \Theta (if it exists). If the size of the input is not obvious, specify what it should be.

1.
```
def matrix_multiply(A, B):
    C = []
    for i in range(len(A)):
        for j in range(len(B[0])):
            C[i][j] = 0
            for k in range(len(A[0])):
                C[i][j] += A[i][k] * B[k][j]
    return C
```

If A is n by m and B is m by k, the runtime is \Theta(n*m*k).

2.
```
def binary_search(L, x):
    if (len(L) == 0): return False
    if x == L[len(L)//2]: return True
    if x < L[len(L)//2]: return binary_search(L[:len(L)//2], x)
    return binary_search(L[len(L)//2:], x)
```

The runtime is \Omega(1), O(\log n).

3.
```
def quicksort(L):
     if (len(L) == 0): return L
    return quicksort([x for x in L if x < L[0]]) +\
           [x for x in L if x == L[0]] +\
           quicksort([x for x in L if x > L[0]])
```

The runtime is \Omega(n) and O(n^2).

The Scheme Programming Language

The Scheme programming language was developed in 1975 as a minimalist dialect of the Lisp programming language. The introductory computer science course of UC-Berkeley (hey, that's us!) used Scheme from 1986 until 2011, remaining the language of choice even through the rise (and sometimes fall) of many other intensely popular programming languages including C++, Perl, Haskell, Java, PHP, and Ruby. It finally ending its 25 year reign in the switch to Python in 2011, Nonetheless, Scheme is a fully-featured language which we'll be implementing an interpretor for over the next few weeks.

If you'd like to learn some more about CS 61A's switch from Scheme, Brian Harvey has a very interesting write up on his website: http://www.cs.berkeley.edu/~bh/61a.html

Scheme Syntax: Prefix Notation!

In Python we use a form of prefix notation when calling functions:

```
>>> foo(1, 2, 3)
>>> mul(2, 3)
>>> add(3, 4)
```

However, Python uses infix notation for several binary operators such as +, -, *, /, “and”, “or”, etc... By contrast, Scheme uses prefix notation for <u>all</u> these operators

```
STk> (+ 2 3)
5
STk> (* 4 3)
12
STk> (and #t #t)
#t
STk> (or #f #t)
#t
```

You can define functions just like you do in Python:

```
STk> (define (fib n)
  (if (or (= n 0) (= n 1))
      1
      (+ (fib (- n 2)) (fib (- n 1)))))
fib
STk> (fib 0)
1
STk> (fib 1)
1
STk> (fib 2)
2
STk> (fib 10)
89
```

You can use high-order functions just like you do in Python:

```
STk> (define (double f n) (* (f n) 2))
double
STk> (double fib 4)
10
STk> (double fib 10)
178
```

You can even use lambda expressions like you do in Python:

```
STk> (double (lambda (x) (* x x)) 20)
800
```

Evironment diagrams even work the same way they do in Python:

```
STk> (define foo (double_func (lambda (x) (* x x))))
foo
STk> (foo 20)
800
STk> (foo 10)
200
```

Exercise: Test your understanding of the above syntax by translating each of the input lines into equivalent Python code.

Defining Procedures

As shown above, you can define functions using the `define` keyword:

```
(define (<procedure name> <arg_1> … <arg_n>)
   <body expression>
)
```

In Scheme, instead of having a sequence of expressions you usually have just a single (though often very complicated) expression for the body of the function.

Conditional Expressions: `if`, and `cond`

Scheme has the usual `if` keyword which looks like a function taking 3 arguments (the conditional argument, and true/false results), but has the usual if behavior of only evaluating an argument if necessary:

```
STk> (if (= 1 1) (* 2 3) (/ 6 0))
6
STk> (if (= 1 0) (* 2 3) (/ 6 0))
*** Error:
```

In addition to the `if` keyword, Scheme also has `cond`, which evaluates conditions in sequence and returns the first value where a condition holds true:

```
STk> (cond ((= 1 0) (/ 6 0))
           ((= 0 1) (/ 5 0))
           ((= 1 1) (* 2 3)))
6
STk> (cond ((= 1 0) (/ 6 0)) ((= 0 1) (/ 5 0)) ((= 1 2) (* 2 3)) (else (+ 1 1)))
2
```

Lists in Scheme

As mentioned above, Scheme is a dialect of LISP, a named derived from “"LISt Processing". So as you might imagine, Scheme has some built-in functionality for dealing with lists:

```
STk> '(1 2 3 4)
(1 2 3 4)
```

Note the quote in front of the opening paren which indicates that this is a **list** and that we are not trying to call the function 1 with arguments 2 3 4. Scheme also has built-in functions `car` and `cdr` which get the first and rest of a list respectively:

```
STk> (car '(1 2 3 4))
1
```

```
STk> (cdr '(1 2 3 4))
(2 3 4)
```

This might look familiar, and that's because it should! Behind the scenes, Scheme actually stores the list (1 2 3 4) as a recursive-list, i.e. as (1 (2 (3 (4)))). Scheme is just smart enough to know that it's easier for humans to read (1 2 3 4) than (1 (2 (3 (4)))), so that's what it prints out. However, if we wanted to construct a list manually, we would use the `cons` function, which works like the `make_rlist` function we've defined in Python:

```
STk> (cons 1 (cons 2 (cons 3 (cons 4 ()))))
(1 2 3 4)
```

Note the use of () at the end to represent the empty list. Finally, we can check to see if a list is empty using the `null?` keyword, i.e.

```
STk> (null? '(1))
#f
STk> (null? ())
#t
```

Exercise: Write a function add_to_end which adds an element to the end of a list:

```
STk> (define (add_to_end L x)

   (if (null? L) (cons x ())

      (cons (car L) (add_to_end (cdr L) x))

   )

)

STk> (add_to_end '(1 2 3 4) 5)

(1 2 3 4 5)
```

Exercise: Write a function in scheme that reverses a list (using the above add_to_end function).

```
(define (reverse_list L)

   (if (null? L) ()

      (add_to_end (reverse (cdr L)) (car L)))

)
```

String Manipulation

Scheme recognizes strings using double-quotes “”.

```
STk> "hello"
"hello"
```

Scheme also has functions first and butfirst (abbreviated bf) which work on strings the way car and cdr work on lists:

```
STk> (first "hello")
h
```

STk> (butfirst "hello")
ello
STk> (bf "hello")
ello

Exercises

1. Define the `map_fn` procedure in Scheme – `map_fn` should take a function name and a list, returning a new list with the function applied to each element of the old list:

```
STk> (define (map_fn f L)

   (if (null? L) ()

        (cons (f (car L)) (map_fn f (cdr L)))

   )

)

STk> (map_fn (lambda (x) (* x x)) '(1 2 3 4))
(1 4 9 16)
```