# Week 9 Discussion

## Orders of growth

It is useful to be able to predict the running time for a function in terms of the parameters it takes. If you have a program that runs too slow to be of any use, the cause could be a function that has an exponential (or otherwise large) order of growth. If you are able to analyze the code and determine where the bottleneck is, you're that much closer to fixing the problem.

To illustrate the effect that order of growth can have, consider the following (Assume a problem of size 1 takes about a microsecond). Imagine you have a function with an order of growth of Θ(2^n), and a function that runs in linear time (Θ(n)). In the time it takes for the first function to run given an input of about 44, you could walk across the U.S. If you ran the same size input on the second function you wouldn't even be able to get out of your seat before the program finishes.

Here is a brief recap of the notation we use to describe orders of growth:

- O(f) is the set of functions that eventually grow no faster than f
- Ω(f) is the set of functions that eventually grow at least as fast as f
- Θ(f) is the set of functions eventually that grows like f

For the following functions, guess the order of growth for an input of size N:

1.
```
def fib(n):
        a, b = 0, 1
        while n > 0:
            a, b = b, a + b
            n -= 1
        return b
```

2.
```
def mod_7(n):
        if n % 7 == 0:
            return 0
        else:
            return 1 + mod_7(n-1)
```

3. What is the order of growth of sum_row? How about sum_matrix?

```
def sum_matrix(m):
        sum = 0
        for x in m:
            for y in x:
                sum += x
        return sum


def make_matrix(n):
        "" Takes an integer n, creates an n by n matrix where the
        value of a m[x][y] is the value of the
```

```
        sum of the previous values in the matrix plus the sum of x
        and y
        """
        entry = 0
        matrix = []
        sum = 0
        for i in range(n):
            matrix.append([])
            for j in range(n):
                matrix[i].append(entry + sum_matrix(matrix))
                entry += 1
        return sum
```

4. What is the order of growth of foo(bar(N)) ?

```
def bar(n):
        if n % 2 == 1:
            return n + 1
        return n

    def foo(n):
        if n < 1:
            return 2
        if n % 2 == 0:
            return foo(n-1) + foo(n-2)
        else:
            return 1 + foo(n-2)
```

**Tree recursion**

Here is a definition of the Tree class you saw in lecture and homework:

```
class Tree:
    """A Tree consists of a label and a sequence
     of 0 or more Trees, called its children."""

    def __init__(self, label, *children):
        """A Tree with given label and children. For convenience, if
         children[k] is not a Tree,
        it is converted into a leaf whose operator is children[k].
         """
        self.__label = label;
        self.__children = [ c if type(c) is Tree else Tree(c) for c
                                in children]

    @property
    def is_leaf(self):
        return not (self[0] or self[1])

    @property
```

```
    def label(self):
        return self.__label

    @property

    def arity(self):
        """The number of my children."""
        return len(self.__children)

    def __iter__(self):
        """An iterator over my children."""
        return iter(self.__children)

    def __getitem__(self, k):
        """My kth child."""
        return self.__children[k]
```

You have also seen binary trees in the form of a binary search tree. A binary tree is a special type of tree whose nodes have no more than two children. What makes a binary search tree special?

Write a function is_binary_search(T) that determines whether or not a tree is a binary search tree (Hint: fill in the function is_binary(T) first).

```
def is_binary_search(T):
 “””Returns True if T is a binary search tree, and False otherwise”””
```

```
def is_binary(T):
    “””Returns True if T is a binary tree, False otherwise”””
```

Binary search trees are most efficient when they are balanced. We call a tree balanced if the depth of the two subtrees of every node never differ by more than 1. Write a function is_balanced that returns True if the given binary tree is balanced, and False otherwise.

```
def is_balanced(T):
```

**Calculator**

We are beginning to dive into the realm of interpreting computer programs. In order to do so, we'll have to examine some new programming languages. The Calculator language, invented for this class, is the first of these examples.

Our Calculator language is actually nearly identical to Python, except it only handles arithmetic operations - add, sub, mul, div, etc. Also unlike Python's arithmetic operators, we'll let Calculator's operators each take an arbitrary number of arguments.

Here's a few examples of Calculator in action:

```
calc> 6
6

calc>mul()
1

calc>add(1, mul(3, sub(3, 7)))
-11
```

Our goal now is to write an interpreter for the Calculator language. The job of an interpreter is, given an expression, evaluate its meaning. So let's talk about expressions.

Representing Expressions

When we type a line at the Calculator prompt and hit enter, we've just sent an expression to the interpreter. We can represent an Expression as an object:

```
class Exp(Object):
    def __init__(self, operator, operands):
        self.operator = operator
        self,operands = operands
    def __repr__(self):
        return 'Exp({0}, {1})'.format(repr(self.operator),
                                      repr(self.operands)))
    def __str__(self):
        operand_strs = ', '.join(map(str, self.operands))
        return '{0}({1})'.format(self.operand, operand_strs)
```

The most important thing to note is that every Exp is represented by its operator and operands. Also another thing to note: an Exp's operands are always themselves Exps as well.

Example: If I wanted to represent the Calculator expression add(2, 3), I would do this by calling the Exp constructor as follows: Exp('add, [2, 3]).

Our Calculator language is only concerned with two kinds of expressions: numbers, which are self-evaluating expressions, and call expressions, which involve an operator acting on some number of arguments, each of which are expressions.

What does it mean for a number to be self-evaluating? If we evaluate an expression that is a number, the value of the expression (i.e. the result returned by evaluating it) is that number. Simple!

What about evaluating call expressions? We can follow this straightforward two-step process.

1. Evaluate the operands.

2. Apply the operator, to the arguments (the values of the operands)
(Sound familiar?)

Using this two-step process, we can interpret any Calculator expression. The first step will be handled by a function called calc_eval, and the second step will be handled by a function called calc_apply.

Here's calc_eval:

```
def calc_eval(exp):
    if type(exp) in (int, float): # if the expression is a number
        return exp
    else:
        #evaluate the operands
        arguments = list(map(calc_eval, exp.operands))
        #apply the function to operands
        return calc_apply(exp.operator, arguments)
```

As you can see, all we've done is follow the rules of evaluation outlined above. If the expression is a number, return it. Else, evaluate the operands and apply the operator the evaluated operands.

How do we apply the operator? With calc_apply:

```
def calc_apply(operator, args):
    if operator in ('add', '+'):
        return sum(args)
    if operator in ('sub', '-'):
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1
argument')
        if len(args) == 1:
            return -args[0]
        return sum(args[0], [-args for args in args[1:]])
    if operator in ('mul', '*'):
        return reduce(mul, args, 1)
```

Depending on what the operator is, we can match it to a corresponding Python call. Each conditional in the function above corresponds to the application of one operator.

Something very important but may not have been obvious: calc_exp deals with expressions, calc_apply deals with values.

**Questions**

Let's say we want to make the variable a contain the object representation of the Calculator expression:

add(4, 5, mul(3, 2)). Fill in the blank:

```
>>>a =
Exp(_____________________________________________________________________)
```

Suppose we typed the following expression into the calc interpreter:

```
calc> add(1, mul(3, sub(3, 7)))
```

How many calls to calc_eval does this generate? How many calls to calc_apply?

The above implementation of the calc_exp and calc_apply does not handle calls to div. Add to them so we can evaluate Calculator div calls:

Example:

```
calc>div(8, 4)
2
```

If a div expression is not given exactly two arguments, you should raise a TypeError.