# Orders of Growth

## Computer Science Mentors 61A

March 7 to March 11, 2016

1. In big-O notation, what is the runtime for `foo`?

   (a)
   ```
   def foo(n):
       for i in range(n):
           print('hello')
   ```

   **Solution:** $O(n)$

   (b) What's the runtime of `foo` if we change `range(n)`:

   i. To `range(n / 2)`?

   **Solution:** $O(n)$

   ii. To `range(10)`?

   **Solution:** $O(1)$

   iii. To `range(10000000)`?

   **Solution:** $O(1)$

2. What is the order of growth in time for the following functions? Use big-O notation.

   (a)
   ```
   def strange_add(n):
       if n == 0:
           return 1
       else:
           return strange_add(n - 1) + strange_add(n - 1)
   ```

**Solution:** $O(2^n)$

(b)
```
def stranger_add(n):
    if n < 3:
        return n
    elif n % 3 ==  0:
        return stranger_add(n - 1) + stranger_add(n - 2) +
           stranger_add(n - 3)
    else:
        return n
```

**Solution:** $O(n)$ is $n$ is a multiple of 3, otherwise $O(1)$.

(c)
```
def waffle(n):
    i = 0
    sum = 0
    while i < n:
        for j in range(50 * n):
            sum += 1
        i += 1
    return sum
```

**Solution:** $O(n^2)$

(d)
```
def belgian_waffle(n):
    i = 0
    sum = 0
    while i < n:
        for j in range(n ** 2):
            sum += 1
        i += 1
    return sum
```

**Solution:** $O(n^3)$

(e)
```
def pancake(n):
    if n ==  0:
        return n
    # Flip will always perform three operations and return
      -n.
    return flip(n) + pancake(n - 1) + pancake(n - 2)
```

**Solution:** $O(2^n)$

(f)
```
def toast(n):
    i = 0
    j = 0
    stack = 0
    while i < n:
        stack += pancake(i)
        i += 1
    while j < n:
        stack += 1
        j += 1
```

```
        return stack
```

**Solution:** $O(n^3)$

3. Consider the following functions:

```
def hailstone(n):
   print(n)
   if n < 2:
    return
   if n % 2 == 0:
        hailstone(n // 2)
   else:
        hailstone((n * 3) + 1)

def fib(n):
   if n < 2:
      return n
   return fib(n - 1) + fib(n - 2)

def slow(n):
    i, j, k = 0, 0, 0
    while i < n:
        while j < n:
            while k < n:
                fib(k)
                k += 1
            fib(j)
            j += 1
        fib(i)
        i += 1

def foo(n, f):
    return n + f(500)
```

In big-O notation, describe the runtime for the following:

(a) `foo(10, hailstone)`

**Solution:** $O(1)$

(b) `foo(3000, fib)`

**Solution:** $O(1)$

(c) `foo(99999999999, slow)`

**Solution:** $O(1)$

4. **Fast Exponentiation:** in this problem, we will examine a real-world algorithm used to improve the speed of calculating exponents.

   (a) First, express the runtime of the naive exponentiation algorithm in big-O notation.

```
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n - 1)
```

   **Solution:** $O(n)$

   (b) Now, express the runtime of the fast exponentiation algorithm in big-O notation.

```
def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n // 2))
    else:
        return b * fast_exp(b, n - 1)
```

   **Solution:** $O(\log n)$

   (c) What about this slightly modified version of `fast_exp`?

```
def fast_exp(b, n):
    for _ in range(50 * n):
        print("Killing time")
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n // 2))
    else:
        return b * fast_exp(b, n - 1)
```

   **Solution:** $O(n \log n)$

5. **Mysterious loops:** What is the order of growth in time for the following functions? Use big-O notation.

(a)
```
def mystery(n):
    for i in range(n):
        while i % 2 != 0:
            print(i)
            i = i - 1
        print("Done")
```

**Solution:** $O(n)$

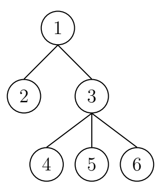
(b)
```
def fun(n):
    for i in range(n):
        for j in range(n * n):
            if j == 4:
                return -1
            print("Fun!")
```

**Solution:** $O(1)$

6. **Orders of Growth and Trees:** Assume we are using the non-mutable Tree implementation introduced in discussion. Consider the following function:

```
def word_finder(t, n, word):
    if label(t) == word:
        n -= 1
        if n == 0:
            return True
    for child in children(t):
        if word_finder(child, n, word) == True:
            return True
    return False
```

(a) What does this function do?
Hint: A path is a sequence of connected nodes. For example, here are four paths in the tree below: $1 \to 2$, $1 \to 3 \to 4$, $1 \to 3 \to 5$, $1 \to 3 \to 6$




**Solution:** This function take a Tree `t`, an integer `n`, and a string `word` in as input. Then, `word_finder` returns True if the word appears as a label in the Tree `n`-times and False otherwise.

(b) If a tree has $n$ total nodes, what is the total runtime for all searches in big-O notation?

**Solution:** $O(n)$

7. **Orders of Growth and Linked Lists:** Consider the following linked list function:

```
def insert_at_end(lst, x):
    if lst.rest is Link.empty:
        lst.rest = Link(x)
    else:
        insert_at_end(lst.rest, x)
```

(a) What does this function do?

> **Solution:** Inserts a value `x` at the end of linked list `lst`.

(b) Say we want to repeatedly insert some numbers into the end of a linked list:

```
def insert_many(lst, n):
    for i in range(n):
        insert_at_end(lst, i)
```

i. Assume `lst` is initially length 1. How long will it take to do the first insertion? The second? The $n$th?

> **Solution:** Notice that the list gets longer with each insertion, so each operation will make it harder to do the next. Therefore, the first insertion will take about 1 unit of time. The second will take about twice as long, at two units of time. The $n$th insertion will take $n$ units of time.

ii. In big-O notation, What is the total runtime to do all the inserts? (total runtime of `insert_many`)

> **Solution:** The total runtime will be the sum of all the inserts: $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} = O(n^2)$