ORDERS OF GROWTH

COMPUTER SCIENCE MENTORS 61A

March 13 to March 17, 2016

1. **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)
```

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

```
def fast_exp(b, n):
    if n == 0:
        return 1
    if n % 2 == 0: # Assume square runs in constant time
        return square(fast_exp(b, n // 2))
    return b * fast_exp(b, n - 1)
```

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

2. **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")
```

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

3. **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 root(t) == word:
        n -= 1
        if n == 0:
            return True
    for branch in branches(t):
        if word_finder(branch, n, word):
            return True
    return False
```

   (a) What does this function do?

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

4. **Orders of Growth and Linked Lists:** For reference, here is our implementation of a Linked List:

```
class Link:
    empty = ... #The empty list, implementation not shown.
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
    def __repr__(self): # implementation not shown
        '''Displays a Link in a more readable manner.
```

Consider the following linked list function:

```
def insert_at_beginning(lst, x):
    return Link(x, lst)
```

(a) What does this function do?

(b) Assume lst is initially length n. How long does it take to do one insert? Two? n?

Now consider:

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

(c) What does this function do?

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

```
def insert_many_end(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 nth?

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

5. **This is a question about Box-and-Pointer diagrams.** Please refer to the Linked List implementation from the previous question.

   (a) Fill in the box and pointer diagram that follows the execution of this code.

```
x = Link(2, Link(5, Link(5)))
y = Link(2, Link(4, Link(6)))
x.rest, y.rest = y.rest, x.rest

def peanutbutter(z):
    if z == Link.empty:
        return Link.empty
    x.first = x.first * z.first
    if z.first % 2 == 0:
        return Link(z.first, peanutbutter(z.rest))
    return Link(Link(x.first))

jelly = peanutbutter(y)
```