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)
```

   Solution: $\Theta(n)$. n decreases by 1 each call, so there are naturally n calls.

   (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)
```

   Solution: $\Theta(\log n)$. n is halved each call, so the number of calls is the number of times n must be halved to get to 1. This is $\log n$.

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

   Solution: $\Theta(n)$. Ignore the constant term. The first call will perform n operations, the second call will perform $n/2$ operations, the third will perform $n/4$ operations, etc. Using geometric series, we see this adds up to $2n$, which is n if we ignore constant terms.

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")
```

   Solution: $\Theta(n)$. The work for when i is divisible by two is constant. Subtracting one will immediately allow us to exit the `while` loop. Therefore, we can concentrate on just the outer loop.

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

   Solution: $\Theta(1)$. Inner loop always immediately exits after running for 4 iterations, independent of n.

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?

Solution: This function take a Tree `t`, an integer `n`, and a string `word` in as input.

Then, `word_finder` returns True if any paths from the root towards the leaves have at least `n` occurrences of the word and False otherwise.

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

Solution: $\Theta(n)$. At worst, we must visit every node of the tree.

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?

Solution: It takes in an existing lst and returns a new list with x at the front.

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

Solution: All inserts will take constant time. No matter how long the list is, it doesn't take any longer to add to the front. One insert will take one unit of time, and two will take roughly twice that. Therefore, the amount of time to do n inserts will be $\Theta(n)$.

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?

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

(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?

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 nth 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_end`)

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

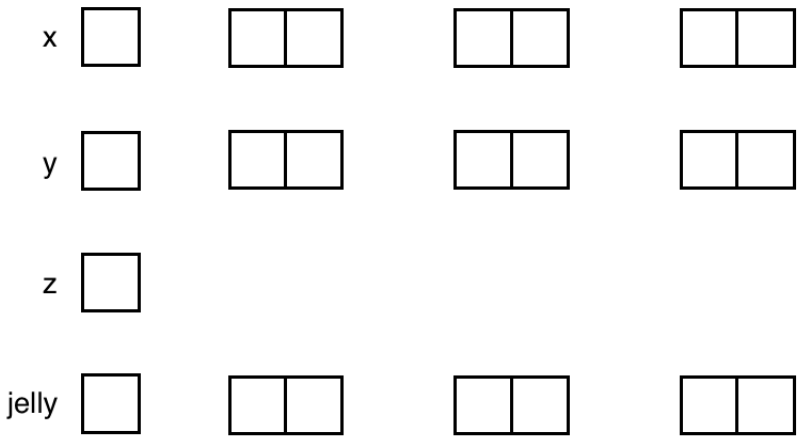
5. **This is a question about Box-and-Pointer diagrams.** Please refer to the Linked List implementation from the previous question.

   (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)
```



Solution: