1 Warmup

What is the order of growth for the following functions?  Answer in terms of $\Theta$ (for example, $\Theta(n)$).

1.1 def fib_iter(n):
    prev, curr, i = 0, 1, 0
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return prev

1.2 def fib_recursive(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib_recursive(n - 1) + fib_recursive(n - 2)

1.3 Write a function that takes in a a linked list and returns the sum of all its elements.
You may assume all elements in lnk are integers.

   def sum_nums(lnk):
   ""
   >>> a = Link(1, Link(6, Link(7)))
   >>> sum_nums(a)
   14
   ""
Orders of Growth & Linked Lists

2 Orders of Growth

When we talk about the efficiency of a function, we are often interested in the following: as the size of the input grows, how does the runtime of the function change? And what do we mean by “runtime”?

- **square(1)** requires one primitive operation: * (multiplication). **square(100)** also requires one. No matter what input $n$ we pass into **square**, it always takes one operation.

<table>
<thead>
<tr>
<th>input</th>
<th>function call</th>
<th>return value</th>
<th>number of operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>square(1)</td>
<td>1 · 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>square(2)</td>
<td>2 · 2</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>square(100)</td>
<td>100 · 100</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$n$</td>
<td>square($n$)</td>
<td>$n · n$</td>
<td>1</td>
</tr>
</tbody>
</table>

- **factorial(1)** requires one multiplication, but **factorial(100)** requires 100 multiplications. As we increase the input size of $n$, the runtime (number of operations) increases linearly proportional to the input.

<table>
<thead>
<tr>
<th>input</th>
<th>function call</th>
<th>return value</th>
<th>number of operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>factorial(1)</td>
<td>1 · 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>factorial(2)</td>
<td>2 · 1 · 1</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>100</td>
<td>factorial(100)</td>
<td>100 · 99 · 1 · 1</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$n$</td>
<td>factorial($n$)</td>
<td>$n · (n - 1) · 1 · 1$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

For expressing complexity, we use what is called big Θ (Theta) notation. For example, if we say the running time of a function `foo` is in Θ($n^2$), we mean that the running time of the process will grow proportionally with the square of the size of the input as it increases to infinity.

- **Ignore lower order terms**: If a function requires $n^3 + 3n^2 + 5n + 10$ operations with a given input $n$, then the runtime of this function is Θ($n^3$). As $n$ gets larger, the lower order terms (10, 5n, and 3$n^2$) all become insignificant compared to $n^3$.

- **Ignore constants**: If a function requires 5$n$ operations with a given input $n$, then the runtime of this function is Θ($n$). We are only concerned with how the runtime grows asymptotically with the input, and since 5$n$ is still asymptotically linear; the constant factor does not make a difference in runtime analysis.

Kinds of Growth

Here are some common orders of growth, ranked from no growth to fastest growth:

- Θ(1) — constant time takes the same amount of time regardless of input size
• $\Theta(\log n)$ — logarithmic time
• $\Theta(n)$ — linear time
• $\Theta(n \log n)$ — linearithmic time
• $\Theta(n^2), \Theta(n^3),$ etc. — polynomial time
• $\Theta(2^n), \Theta(3^n),$ etc. — exponential time (considered “intractable”; these are really, really horrible)

In addition, some programs will never terminate if they get stuck in an infinite loop.

**Questions**

What is the order of growth for the following functions?

2.1 `def sum_of_factorial(n):
    if n == 0:
        return 1
    else:
        return factorial(n) + sum_of_factorial(n - 1)`

2.2 `def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total`

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

2.4 `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)`

What is the order of growth of $\text{foo}(\text{bar}(n))$?
3 Linked Lists

There are many different implementations of sequences in Python. Today, we'll explore the linked list implementation.

A linked list is either an empty linked list, or a Link object containing a first value and the rest of the linked list.

To check if a linked list is an empty linked list, compare it against the class attribute `Link.empty`:

```python
if link is Link.empty:
    print('This linked list is empty!')
else:
    print('This linked list is not empty!')
```

**Implementation**

```python
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest_str = ', ' + repr(self.rest)
        else:
            rest_str = ''
        return 'Link({0}{1}).format(repr(self.first), rest_str)

@property
def second(self):
    return self.rest.first

@second.setter
def second(self, value):
    self.rest.first = value

def __str__(self):
    string = '<'
    while self.rest is not Link.empty:
        string += str(self.first) + ' ' + str(self.rest)
        self = self.rest
    return string + str(self.first) + '>'
```
Questions

3.1 Write a function that takes in a Python list of linked lists and multiplies them element-wise. It should return a new linked list.

If not all of the Link objects are of equal length, return a linked list whose length is that of the shortest linked list given. You may assume the Link objects are shallow linked lists, and that lst_of_lnks contains at least one linked list.

```python
def multiply_lnks(lst_of_lnks):
    """
    >>> a = Link(2, Link(3, Link(5)))
    >>> b = Link(6, Link(4, Link(2)))
    >>> c = Link(4, Link(1, Link(0, Link(2))))
    >>> p = multiply_lnks([a, b, c])
    >>> p.first
    48
    >>> p.rest.first
    12
    >>> p.rest.rest.rest
    ()
    """
```

3.2 Write a function that takes a sorted linked list of integers and mutates it so that all duplicates are removed.

```python
def remove_duplicates(lnk):
    """
    >>> lnk = Link(1, Link(1, Link(1, Link(1, Link(1, Link(5))))))
    >>> unique = remove_duplicates(lnk)
    >>> unique
    Link(1, Link(5))
    >>> lnk
    Link(1, Link(5))
    """
```
4 Midterm Review

4.1 Write a function that takes a list and returns a new list that keeps only the even-indexed elements of lst and multiplies them by their corresponding index.

```python
def even_weighted(lst):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> even_weighted(x)
    [0, 6, 20]
    """
    return [_________________________________________________

4.2 The quicksort sorting algorithm is an efficient and commonly used algorithm to order the elements of a list. We choose one element of the list to be the pivot element and partition the remaining elements into two lists: one of elements less than the pivot and one of elements greater than the pivot. We recursively sort the two lists, which gives us a sorted list of all the elements less than the pivot and all the elements greater than the pivot, which we can then combine with the pivot for a completely sorted list.

First, implement the quicksort_list function. Choose the first element of the list as the pivot. You may assume that all elements are distinct.

```python
def quicksort_list(lst):
    """
    >>> quicksort_list([3, 1, 4])
    [1, 3, 4]
    """
    if ____________________________________________________:
        __________________________________________________
        pivot = lst[0]

        less = ______________________________________________

        greater = ____________________________________________

        return _____________________________________________
```
4.3 Write a function that takes in a list and returns the maximum product that can be formed using nonconsecutive elements of the list. The input list will contain only numbers greater than or equal to 1.

```python
def max_product(lst):
    """Return the maximum product that can be formed using lst without using any consecutive numbers
    >>> max_product([10,3,1,9,2]) # 10 * 9
    90
    >>> max_product([5,10,5,10,5]) # 5 * 5 * 5
    125
    >>> max_product([])
    1
    """
```

4.4 An expression tree is a tree that contains a function for each non-leaf node, which can be either '+' or '*'. All leaves are numbers. Implement `eval_tree`, which evaluates an expression tree to its value. You may want to use the functions `sum` and `prod`, which take a list of numbers and compute the sum and product respectively.

```python
def eval_tree(tree):
    """Evaluates an expression tree with functions the root.
    >>> eval_tree(tree(1))
    1
    >>> expr = tree('*', [tree(2), tree(3)])
    >>> eval_tree(expr)
    6
    >>> eval_tree(tree('+', [expr, tree(4), tree(5)]))
    15
    """
```
Complete redundant_map, which takes a tree t and a function f, and applies f to the node \(2^d\) times, where d is the depth of the node. The root has a depth of 0.

```python
def redundant_map(t, f):
    ""
    >>> double = lambda x: x*2
    >>> tree = Tree(1, [Tree(1), Tree(2, [Tree(1, [Tree(1)])])])
    >>> print_levels(redundant_map(tree, double))
    [2] # 1 * 2 \(^1\); Apply double one time
    [4, 8] # 1 * 2 \(^2\), 2 * 2 \(^2\); Apply double two times
    [16] # 1 * 2 \(^2\ * 2\); Apply double four times
    [256] # 1 * 2 \(^2\ * 3\); Apply double eight times
    ""
    t.label = _________________________________________________
    
    new_f = ___________________________________________________
    
    t.branches = ______________________________________________
    
    return t
```