1 Time and 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 \cdot 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>square(2)</td>
<td>2 \cdot 2</td>
<td>1</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>100</td>
<td>square(100)</td>
<td>100 \cdot 100</td>
<td>1</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>( n )</td>
<td>\text{square}(n)</td>
<td>( n \cdot 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 \cdot 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>factorial(2)</td>
<td>2 \cdot 1 \cdot 1</td>
<td>2</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>100</td>
<td>factorial(100)</td>
<td>100 \cdot 99 \cdots 1 \cdot 1</td>
<td>100</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>( n )</td>
<td>\text{factorial}(n)</td>
<td>( n \cdot (n - 1) \cdots 1 \cdot 1 )</td>
<td>( n )</td>
</tr>
</tbody>
</table>

**Big Theta Notation**

For expressing complexity, we use what is called big \( \Theta \) (Theta) notation. For example, if we say the running time of a function *foo* is in \( \Theta(n^2) \), 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 \( \Theta(n^3) \). As \( n \) gets larger, the lower order terms (10, 5\( n \), and 3\( n^2 \)) all become insignificant compared to \( n^3 \).

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

### Orders of Growth

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

- \( \Theta(1) \) — 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
- \( \Theta(n^2), \Theta(n^3), \text{etc.} \) — polynomial time
- \( \Theta(2^n), \Theta(3^n), \text{etc.} \) — exponential time (considered “intractable”; these are really, really horrible)

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

What is the order of growth for the following functions?

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

\( \Theta(n^2) \), we will call factorial \( n \) times with arguments \( n, n - 1, n - 2, ..., 0 \). The sum from 0 to \( n \) is approximately \( n^2 \).

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

\( \Theta(\log(n)) \), because our while loop iterates at most \( \log(n) \) times, due to \( n \) being halved in every iteration.
1.3  
```python
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

Θ(1), since at worst it will require 6 recursive calls to reach the base case. So this is Θ(6), which can be reduced to Θ(1).

1.4  
```python
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 foo(bar(n))?  
Θ(n²)

## 2  Mutable Data

Let’s imagine you order a mushroom and cheese pizza from La Val’s, and that they represent your order as a list.

A couple minutes later, you realize that you really want onions on the pizza. Based on what we know so far, La Val’s would have to build an entirely new list to add onions:

```python
>>> pizza2 = pizza1 + ['onions']  # creates a new python list
>>> pizza2
['cheese', 'mushrooms', 'onions']
>>> pizza1  # the original list is unmodified
['cheese', 'mushrooms']
```

But this is silly, considering that all La Val’s had to do was add onions on top of pizza1 instead of making an entirely new pizza2.

Python actually allows you to *mutate* some objects, including lists and dictionaries. Mutability means that the object’s contents can be changed. So instead of building a new pizza2, we can use pizza1.append('onions') to mutate pizza1.

```python
>>> pizza1.append('onions')
>>> pizza1
['cheese', 'mushrooms', 'onions']
```
Although lists and dictionaries are mutable, many other objects, such as numeric types, tuples, and strings, are **immutable**, meaning they cannot be changed once they are created. We can use the familiar indexing operator to mutate a single element in a list. For instance `lst[4] = 'hello'` would change the fifth element in `lst` to be the string 'hello'. In addition to the indexing operator, lists have many mutating methods. List **methods** are functions that are bound to a specific list. Some useful list methods are listed here:

1. **append**(el) adds el to the end of the list
2. **insert**(i, el) insert el at index i (does not replace element but adds a new one)
3. **remove**(el) removes the first occurrence of el in list, otherwise errors
4. **pop**(i) removes and returns the element at index i

```python
>>> sharks = ['joe thornton',
            'patrick marleau']
>>> sharks.append('logan couture')
>>> sharks.pop(1)
'patrick marleau'
>>> sharks
['joe thornton', 'logan couture']
```
2.1 Consider the following definitions and assignments and determine what Python would output for each of the calls below if they were evaluated in order. It may be helpful to draw the box and pointers diagrams to the right in order to keep track of the state.

(a) ```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
>>> lst1 == lst2  # compares each value
True
```  
(b) ```
>>> lst1 == lst2  # compares references
False
```  
(c) ```
>>> lst2 = lst1
>>> lst2 is lst1
True
```  
(d) ```
>>> lst1.append(4)
>>> lst1
[1, 2, 3, 4]
```  
(e) ```
>>> lst2
[1, 2, 3, 4]
```  
(f) ```
>>> lst2[1] = 42
>>> lst2
[1, 42, 3, 4]
```  
(g) ```
>>> lst1 = lst1 + [5]
>>> lst1 == lst2
False
```  
(h) ```
>>> lst1
[1, 42, 3, 4, 5]
```  
(i) ```
>>> lst2
[1, 42, 3, 4]
```  
(j) ```
>>> lst2 is lst1
False
```
2.2 Write a function `square_elements` which takes a `lst` and replaces each element with the square of that element. *Mutate* `lst` *rather than returning a new list.*

```python
def square_elements(lst):
    ""
    >>> lst = [1, 2, 3]
    >>> square_elements(lst)
    >>> lst
    [1, 4, 9]
    ""
    for i in range(len(lst)):
        lst[i] = lst[i]**2
```

2.3 Write a function that takes in two values `x` and `el`, and a list, and adds as many `el`'s to the end of the list as there are `x`'s.

```python
def add_this_many(x, el, lst):
    """ Adds el to the end of lst the number of times x occurs in lst."
    >>> lst = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5]
    >>> add_this_many(2, 2, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5, 2, 2]
    """
    count = 0
    for element in lst:
        if element == x:
            count += 1
    while count > 0:
        lst.append(el)
        count -= 1
```

2.4 Write a function that rotates the elements of a list to the right by `k`. Elements should not “fall off”; they should wrap around the beginning of the list. `rotate` should return a new list. To make a list of `n` 0's, you can do this: `[0] * n`

```python
def rotate(lst, k):
    """ Return a new list, with the same elements of lst, rotated to the right k."
    >>> x = [1, 2, 3, 4, 5]
    >>> rotate(x, 3)
    [3, 4, 5, 1, 2]
    """
    n = len(lst)
    ret = [0] * n
```
for i in range(n):
    j = (i + k) % n
    ret[j] = lst[i]
return ret

or

return lst[-k:] + lst[:-k]

Environment Diagrams

2.5 Fill in the second line below so that s evaluates to the list shown in the diagram. 
You may only use brackets, commas, and s in your answer.

>>> s = [1, 2]

>>> s = ________________________________________________________

>>> s = [1, 2]

>>> s = [s, [s]]

2.6 Draw the box and pointer diagram that results from executing the code below until 
the entire program is finished or an error occurs:

w = [2, 2]
z = [2, 2, w]
x = z[w[0]]
y = list(x)
x.append([w.pop()])
y, x = z[2], y
w[0] = z.pop()
3 Nonlocal

Until now, you’ve been able to access variables in parent frames, but you have not been able to modify them. The `nonlocal` keyword can be used to modify a variable in the parent frame outside the current frame. For example, consider `stepper`, which uses `nonlocal` to modify `num`:

```python
def stepper(num):
    def step():
        nonlocal num  # declares num as a nonlocal variable
        num = num + 1  # modifies num in the stepper frame
        return num
    return step
```

However, there are two important caveats with `nonlocal` variables:

- **Global variables** cannot be modified using the `nonlocal` keyword.
- **Variables in the current frame** cannot be overridden using the `nonlocal` keyword. This means we cannot have both a local and nonlocal variable with the same names in a single frame.

Some Common Misconceptions

3.1 What is wrong with the following code?

```python
a = 5
def another_add_one():
    nonlocal a
    a += 1
another_add_one()
```

Nonlocal cannot be used to modify variables in the global frame.

3.2 What is wrong with the following code?

```python
def adder(x):
    def add(y):
        nonlocal x, y
        x += y
    return x
    return add
adder(2)(3)
```

There is no variable `y` defined in a parent frame, and `y` is already a local variable.
Fill in the Blank

3.1 The bathtub below simulates an epic battle between Finn and Kylo Ren over a populace of rubber duckies. Fill in the body of ducky so that all doctests pass.

def bathtub(n):
    ""
    >>> annihilator = bathtub(500) # the force awakens...
    >>> kylo_ren = annihilator(10)
    >>> kylo_ren()
    490 rubber duckies left
    >>> rey = annihilator(-20)
    >>> rey()
    510 rubber duckies left
    >>> kylo_ren()
    500 rubber duckies left
    ""
    def ducky_annihilator(rate):
        def ducky():

            nonlocal n
            n = n - rate
            print(n, 'rubber duckies left')

        return ducky
    return ducky_annihilator