# Discussion 6

DSC 20, Spring 2023

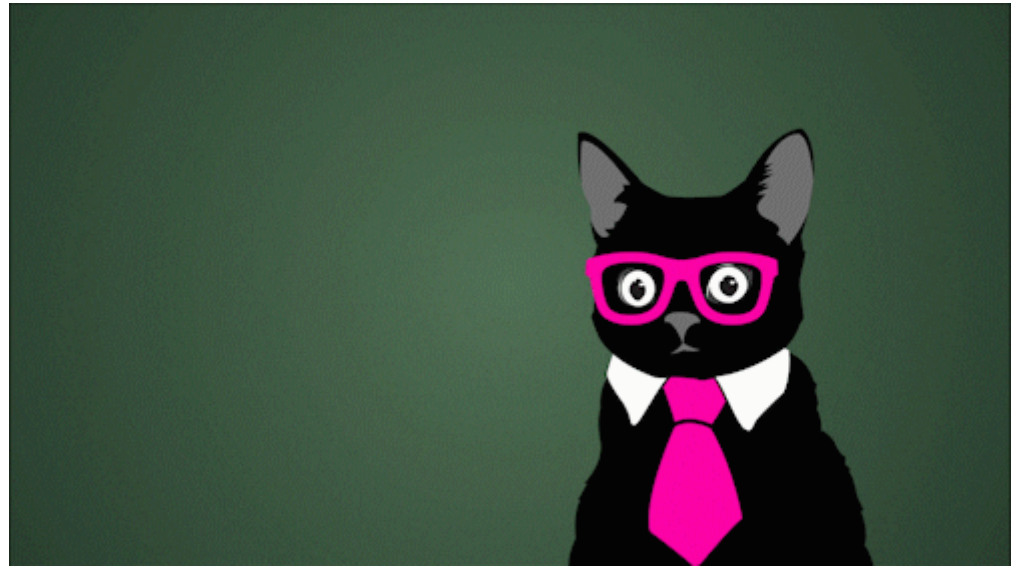
# Recursion, Complexity

# Agenda

- **Complexity** - Math Review, Interpretation, Calculation
- **Recursion** - Base Case, Recursive Calls, Logic

# Why Complexity?

Time complexity is an empirical method to measure the efficiency of code. Since everyone's computer is different, we can't compare code with actual numbers. Instead, we classify them into different runtime categories that allow us to infer its runtime relative to our input.



# Complexity Fundamentals

1. Code should be quantified into big O values
2. Nested code will have compounded big O values
3. The largest term dictates the growth rate (i.e. only largest term matters)
4. Constants are irrelevant
5. big O analysis uses similar ideas to calculus and limits - reference your math knowledge

# Complexity - Basic Interpretation

In [ ]:
```python
1 + 1 # O(1)
```

In [ ]:
```python
for _ in range(n): # O(n)
    print(n)
```

In [ ]:
```python
for i in range(x): # O(n^2)
    for j in range(y):
        print(i + j)
```

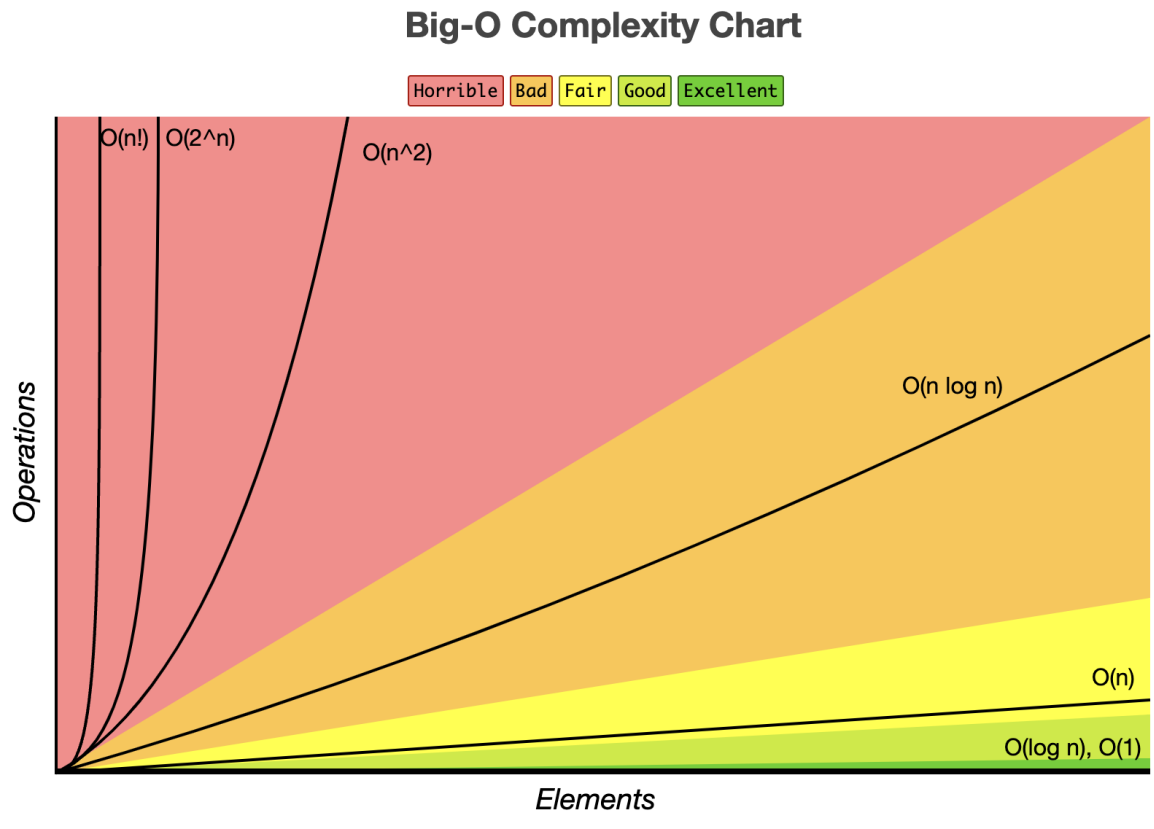
In [ ]:
```python
i = n
while i > 0: # O(log(n))
    i = i // 2
    print(i)
```

**note:** These are very basic examples. Just because there's a nested for loop does not necessarily mean that the runtime is O(n^2). Runtime depends on the number of iterations happening and the cost of each calculation. For example, every function you've used in this class so far has a specific runtime (reference documentation)

# Complexity - Calculation

**tips**:

1. Always look for "hallmark features" (ex. if I see i // 2, there's probably log involved)
2. If there are loops, check the iteration count (don't assume n runtime)
3. Order of Growth follows mathematical principles. Only consider the largest term (without constants)


Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

source

# Complexity Checkpoint

Determine the time complexity of these 3 functions

# Complexity Question 1

In [7]:
```
def count_vowels(string):
    """
    function that takes a string as input
    and returns the number of vowels in the string.
    """
    count = 0
    for char in string:
        if char in "aeiouAEIOU":
            count += 1
    return count
```

# Complexity Question 1

In [7]:
```python
def count_vowels(string):
    """
    function that takes a string as input
    and returns the number of vowels in the string.
    """
    count = 0
    for char in string:
        if char in "aeiouAEIOU":
            count += 1
    return count
```

**Solution:** O(n)

Simple for loop, which runs for n times (n being the length of the input string). WIthin the loop, the operations that happen are all O(1), which yields the resulting expression of:

O(count_vowels) = n * 1 = O(n)

# Complexity Question 2

In [9]:

```
def bubble_sort(arr):
    """
    function that sorts a list of values.
    """
    n = len(arr)
    for i in range(n):
        for j in range(n - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

# Complexity Question 2

In [9]:
```python
def bubble_sort(arr):
    """
    function that sorts a list of values.
    """
    n = len(arr)
    for i in range(n):
        for j in range(n - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

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

Simple for loop, which runs for n times (n being the length of the input list). WIthin the loop, there's a second for loop who runs for n-1 times. Within these 2 for loops, operations are all constant. This yields the resulting expression of:

O(most_frequent_numbers) = n*(n-1)*1 = n^2 - n = O(n^2)

# Complexity Question 3

In [11]:

```
def most_frequent_numbers(arr):
    """
    function to find the two numbers that
    appear most frequently in the list.
    """
    nums_dict = {}
    for num in arr:
        if num in nums_dict:
            nums_dict[num] += 1
        else:
            nums_dict[num] = 1
    most_frequent = sorted(nums_dict.items(), \
           key=lambda x: x[1], reverse=True)[:2]
    return [num[0] for num in most_frequent]
```

# Complexity Question 3

In [11]:
```
def most_frequent_numbers(arr):
    """
    function to find the two numbers that
    appear most frequently in the list.
    """
    nums_dict = {}
    for num in arr:
        if num in nums_dict:
            nums_dict[num] += 1
        else:
            nums_dict[num] = 1
    most_frequent = sorted(nums_dict.items(), \
           key=lambda x: x[1], reverse=True)[:2]
    return [num[0] for num in most_frequent]
```

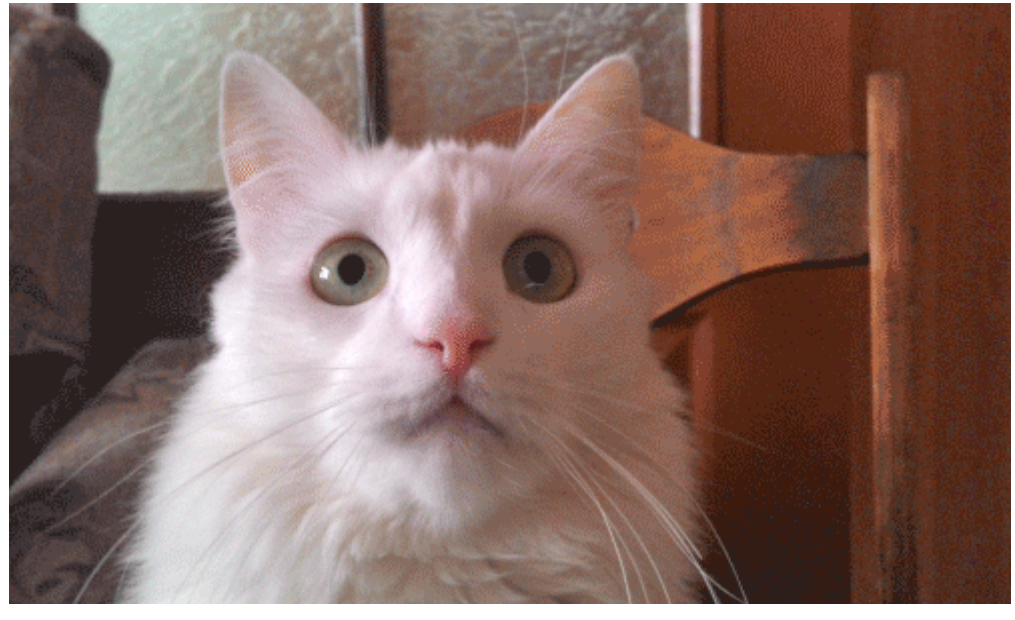
**Solution:** O(nlog(n))

Simple for loop, which runs for n times (n being the length of the input list). WIthin the loop, the operations that happen are all O(1). After the loop, sorted() is called, which is a sorting algorithm with worst case time complexity of O(nlogn). This yields the resulting expression of:

O(most_frequent_numbers) = n*1 + nlog(n) = O(nlog(n))

# Why Recursion?

Recursion is a design method for code. A lot of important ideas' optimal solutions are recursive and many algorithms depend on recursion to function correctly (ex. BFS, DFS, Dijkstra's algorithm, BST's). You can't escape it :)



# Recursion - Base Case

Base case(s) are regarded as the most important part of a recursive function. They determine the stop point for recursion and begin the argument passing up the "stack" of recursive calls. Without a well written base case, recursion will either never end or end incorrectly. When writing recursion questions, always start with determining the base case.

# Recursion - Recursive Calls

The crux of a recursive function working is the recursive calls. These calls will repeat until the base case is reached, creating the "stack" of recursive calls that will begin resolving at the base case. Keep in mind when writing recursive calls that every call needs to trend towards the base case.

# Recursion - Example

In [16]:
```
def list_product(lst):
    """
    recursive function to find the product of every
    element in a list. Discuss recursive structure
    """
    if len(lst)==0:
        return 1
    else:
        return list_product(lst[1:]) * lst[0]
```

In [17]:
```
list_product([1,2,3])
```

Out[17]:
```
6
```

# Recursion - Tracing Logic


The recurisve calls of the function generates a "stack" of recursive functions to resolve, each waiting for the result from the next until it can be solved. No resolution can happen until the base case is reached and returns the iniital value.

# Recursion - Bad Base Case

In [6]:
```
def list_product_wrong(lst):
    if len(lst)==1:
        return 1
    return list_product_wrong(lst[1:]) * lst[0]
list_product_wrong([1,2,3])
```

Out[6]: 2

In [4]:
```
def list_product_wrong(lst):
    if len(lst)==-1:
        return 1
    return list_product_wrong(lst[1:]) * lst[0]
list_product_wrong([1,2,3])
```

```
---------------------------------------------------------------
-----------
RecursionError                                Traceback (most recent
call last)
Cell In[4], line 5
      3         return 1
      4     return list_product_wrong(lst[1:]) * lst[0]
----> 5 list_product_wrong([1,2,3])

Cell In[4], line 4, in list_product_wrong(lst)
      2 if len(lst)==-1:
      3     return 1
```

```
----> 4 return list_product_wrong(lst[1:]) * lst[0]

Cell In[4], line 4, in list_product_wrong(lst)
      2 if len(lst)==-1:
      3     return 1
----> 4 return list_product_wrong(lst[1:]) * lst[0]

    [... skipping similar frames: list_product_wrong at line 4
(2969 times)]

Cell In[4], line 4, in list_product_wrong(lst)
      2 if len(lst)==-1:
      3     return 1
----> 4 return list_product_wrong(lst[1:]) * lst[0]

Cell In[4], line 2, in list_product_wrong(lst)
      1 def list_product_wrong(lst):
----> 2     if len(lst)==-1:
      3         return 1
      4     return list_product_wrong(lst[1:]) * lst[0]

RecursionError: maximum recursion depth exceeded while calling a
Python object
```

# Recursion Checkpoint

Write a recursive function that checks if a string is a palindrome.

In [39]:
```
def check(word):
    """
    Recursive function to check if a string
    is a palindrome (same word reversed).

    args:
        word (string): string to be considered

    returns:
        True if it's a palindrome, false otherwise.

    >>> is_palindrome('racecar')
    True
    >>> is_palindrome('chatgpt')
    False
    """
    # Write your implementation here
```

# Checkpoint Solution

In [41]:
```
def check(word):
    if len(word) <=1:
        return True
    else:
        return all([check(word[1:-1]), word[0] == word[-1]])
```

In [42]:
```
check('racecar')
```

Out[42]: True

In [43]:
```
check('chatgpt')
```

Out[43]: False

# Some Practice Questions

# Question 1

Write a recursive function to reverse a string.

In [13]:
```
def reverse_recursive(s):
    """
    Recursive function to reverse a string

    args:
        s(string): string to be reversed
    returns:
        reversed string

    >>> reverse_recursive('essirahc')
    charisse
    """
    # your implementation here
```

# Question 1 Solution

In [14]:
```
def reverse_recursive(s):
    if len(s) == 0:
        return s
    else:
        return s[-1] + reverse_recursive(s[:-1])

reverse_recursive('essirahc')
```

Out[14]:
```
'charisse'
```

# Question 2

Write a recursive function that flattens a nested list.

In [ ]:

```
def flatten(lst):
    """
    Recursive function to flatten a nested list

    args:
        lst (list): nested list to flatten
    returns:
        flattened list

    >>> flatten([[1,2], [3,4], [5],6,7,8])
    [1,2,3,4,5,6,7,8]
    """
    # your implementation here
```

# Question 2 Solution

In [25]:
```
def flatten(lst):
    if len(lst)==0:
        return []
    else:
        if isinstance(lst[0],list):
            return flatten(lst[0]) + flatten(lst[1:])
        else:
            return [lst[0]] + flatten(lst[1:])
flatten([[1,2], [3,4], [5],6,7,8])
```

Out[25]:
```
[1, 2, 3, 4, 5, 6, 7, 8]
```

# Question 3

What's the time complexity of the following function?

In [30]:
```
def complexity(x):
    output = []
    for i in range(x + 1):
        for j in range(i):
            output.append((i,j))
    return output
complexity(3)
```

Out[30]:
```
[(1, 0), (2, 0), (2, 1), (3, 0), (3, 1), (3, 2)]
```

# Question 3 Solution

The outer loop runs for n+1 iterations (where n is the input number x). The inner loop depends on the value of the outer loop. The operation(s) inside of the loops are linear. Since the inner loop depends on the outer loop, it's not as simple as multiplying 2 values together. We have to determine the number of runs for the loops together. A quick analysis shows that the loops will run 1 + 2 + 3 + ... + (n-1) + n times, which (recall from math: summation formulas) is equivalent to the expression:

O(complexity) = (n(n+1) / 2) * 1 = (n^2 + n) / 2 = **O(n^2)**

# Question 4

What's the time complexity of the following function?

In [ ]:

```
def complexity(x):
    total = 1
    k_sum = 0

    for i in range(x):
        for j in range(x**2, x**2 + 5*x - 50):
            total += j
        k = x
        while k > 0:
            k = k // 2
            k_sum += k

    return total, k_sum
```

# Question 4

What's the time complexity of the following function?

In [ ]:
```
def complexity(x):
    total = 1
    k_sum = 0

    for i in range(x):
        for j in range(x**2, x**2 + 5*x - 50):
            total += j
        k = x
        while k > 0:
            k = k // 2
            k_sum += k

    return total, k_sum
```

# Question 4 Solution

The outer loop runs for n iterations (where n is the input number x). To find the time complexity of the inner for loop, we can subtract the starting value from the stopping value: (n**2 + 5*n - 50) - n**2 = 5*n - 50. Then, for the inner while loop, since we are dividing k in half every iteration, the while loop will have a time complexity of logn. In order to calculate

the final time complexity, we multiply the outer loop's time complexity with the inner loops', resulting in:

O(complexity) = n((5n - 50) + logn) = 5n**2 - 50n + nlogn = **O(n^2)**

*__Note__: remember n^2 > nlogn

# Thanks for coming!

There's a discussion quiz on canvas!