Day - 01 (July 22)

Logarithms & Exponents

Exponent	Logarithms
2 ^ x = y	log base 2 x = y
2 ^ = 4	log base 2 of 4 = 2

ch: Math

11.Factorials

```
In mathematics the factorial of a non-negative integer n, divided by n!, is
this the product of all positive integer less than or equal to n. The
Factorial of  n also equals the product of n with the next smaller
factorial.
```

Formula:

```
n! = n x (n-1) x (n-2) x (n-3) x .... x 3 x 2 x 1
```

The value of 0! is 1. according to the convention for an empty product.

#The Result of the factorial grow even faster than exponentiation

n! grows faster than 2 ^ n

	n!	2 ^ n
2	2	4
3	6	4
4	24	8
5	120	16
6	720	32

Assigment - Factorial

Solution Using for loop:

```
def new_possible_orders(num_posts):
        fac = 1
        for i in range(1, num_posts + 1):
                fac = fac * i
        return fac
```

Solution using recursion method:

```
def num_possible_orders(num_posts):
	if num_posts == 1: # i have slight doubt here verify boot.dev website
		return num_posts
	else:
		return num_posts * num_possible_orders(num_posts - 1)
```

C1 : Exponential Decay

```
	In Physics, exponential decay is a process where a quantity decreases overtime at a rate proportional to its current value.
```

Scenario

```
we have found that instagram influencers tend to lose followers similarly. This means that the more followers you have, the faster you lose them.
```

Assignment

```
complete the decayed_followers function.

if calculates the final value of a quantity after a certain time has passed given its initial value and reate of decay.
The retention_rate is the opposite of fractional_lost_daily if an influencer lost 0.2(or 20%) of their followers each day, then the retention rate would be 0.8(or 80%)
```

#solution

```
def decayed_followers(intl_followers, fraction_lost_daily, days):
    res = intl_followers * (1 - fraction_lost_daily) ** days
    return res
```

Logarithmic Scale

In some cases, data can span several orders of magnitude, making it difficult to visualize on a linear scale. A logarithmic scale can help by compressing the data so that it's easier to understand.

For example, at Socialytics we have influencers with follower counts ranging from 1 to 1,000,000,000. If we want to plot the follower count of each influencer on a graph, it would be difficult to see the differences between the smaller follower counts. We can use a logarithmic scale to compress the data so that it's easier to visualize.

Assignment

Write a function `log_scale(data, base)` that takes a list of positive numbers `data`, and a logarithmic `base`, and returns a new list with the logarithm of each number in the original list, using the given base.

Example

```
# Output: [0.0, 1.0, 2.0, 3.0]

log_scale([1, 2, 4, 8], 2)
# Output: [0.0, 1.0, 2.0, 3.0]
```

#solution

```
def log_scale(data, base):
    arr = []
    for el in data:
        arr.append(math.log(el, base))

    return arr
```

----------- End of Math Chapter ------------#

Chapter 3 - Polynomial Time

1.Big O Notation

There are *lots* of existing algorithms; some are fast and some are slow. Some use lots of memory. It can be hard to decide which algorithm is the best to solve a particular problem. ["Big O"]() analysis (pronounced "Big Oh", not "Big Zero") is one way to compare algorithms.

> Big O is a characterization of algorithms according to their worst-case growth rates

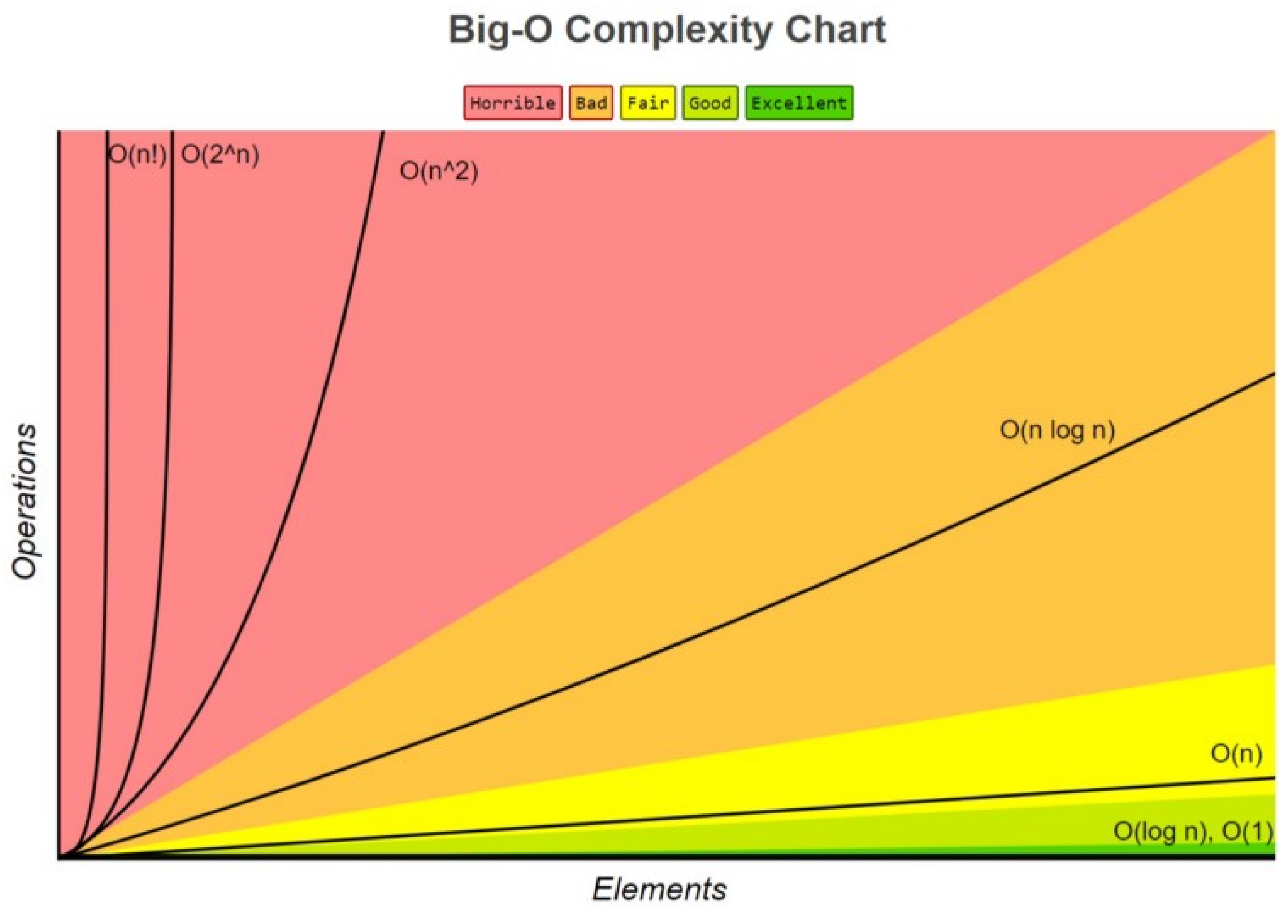
We write Big-O notation like this:

```
O(formula)
```

Where `formula` describes how an algorithm's run time or space requirements grow **as the input size grows.**

- `O(1)` - constant
- `O(n)` - linear
- `O(n^2)` - squared
- `O(2^n)` - exponential
- `O(n!)` - factorial

The following chart shows the growth rate of several different Big O categories. The size of the input is shown on the `x axis` and how long the algorithm will take to complete is shown on the `y axis`.



[- source]

As you can see, as the size of inputs grows, the time they take to complete generally increases. The *rate* at which the functions become slower is defined by their Big O category.

For example, `O(n)` algorithms slow down more slowly than `O(n^2)` algorithms.

2.O(n) - Order "n"

`O(n)` is *very* common - Any algorithm which has its number of steps grow at the same rate as its input size is classified as `O(n)`

For example, our `find min` algorithm from earlier is `O(n)`:

1. Set `min` to positive infinity.
2. For each number in the list, compare it to `min`. If it is smaller, set `min` to that number.
3. `min` is now set to the smallest number in the list.

The input to the `find min` algorithm is a list of size `n`. Because we loop over each item in the input once, we add one step to our algorithm for each item in our list.

As we use `find min` with larger and larger inputs, the length of time it takes to execute the function grows at a steady linear pace. We can reasonably estimate the time it will take to run, based on a previous measurement:

```
# if we find that...
find_min(10 items) = 2 milliseconds

# ...then we can estimate
find_min(100 items)  =  20 milliseconds
find_min(1000 items) = 200 milliseconds
```

Assignment

At Socialytics we now need to display to our users the people who follow them with the *highest* engagement count. This will help them know which of their followers they should follow back.

Complete the `find_max` function. It should take a list of integers and return the largest value in the list.

The "runtime complexity" (aka Big O) of this function should be `O(n)`

#solution

```
def find_max(nums):
    if len(nums) == 0:
        return None

    max_num = nums[0]
    for num in nums:
        if num > max_num:
            max_num = num
    return max_num
```

3.O(n^2) - Order "n squared"

`O(n^2)` grows in complexity much more rapidly. For small and medium input sizes, these algorithms can still be very useful.

A common way that an algorithm will fall into the `O(n^2)` class is by using a nested loop, where the number of iterations of each loop is equal to the number of items in the input.

Assignment

Socialytics needs search capabilities! For now, we'll build something slow but simple. Our users want to give us the full name of a fellow Instagrammer, and we need to find them by checking if their first and last names exist in our database.

Complete the `does_name_exist` function. It should loop over all of the first/last name combinations in the `first_names` and `last_names` lists. If it finds a combination that matches the `full_name` it should return `True`. If the full name isn't found, it should return `False` instead.

Observe

When you run your completed code, notice how each successive call to `does_name_exist` takes quite a bit longer. Assuming the length of `first_names` and `last_names` is the same, each new name doesn't add `n` steps to the algorithm it adds `n^2` steps.

If `does_name_exist(10 first and last names)` takes just `1` second to complete, then we can assume the following timings are roughly accurate:

- `does_name_exist(100 first and last names)` = 100 seconds

- `does_name_exist(1,000 first and last names)` = 10,000 seconds
- `does_name_exist(10,000 first and last names)` = 1,000,000 seconds

#solution

```
def does_name_exist(first_names, last_names, full_name):
    for first_name in first_names:
        for last_name in last_names:
            if first_name + ' ' + last_name == full_name:
                return True
    return False
```

N^2 Quiz

Refer to the following functions, and assume that `first_names` and `last_names` are the same length.

```python
def print_names_one(first_names, last_names):
    for first_name in first_names:
        print(first_name)
    for last_name in last_names:
        print(last_name)
```

```python
def print_names_two(first_names, last_names):
    for first_name in first_names:
        for last_name in last_names:
            print(first_name, last_name)
```

What are the Big O complexities of print_names_one and print_names_two respectively?
#answer = O(n), O(n ^ 2)

N^2 Quiz

Refer to the following functions, and assume that `first_names` and `last_names` are the same length.

```python
def print_names_one(first_names, last_names):
    for first_name in first_names:
        print(first_name)
    for last_name in last_names:
        print(last_name)
```

```python
def print_names_two(first_names, last_names):
    for first_name in first_names:
        for last_name in last_names:
            print(first_name, last_name)
```

Which function will finish faster?
#answer = print_names_one

above 12.30 PM:

- Refactoring the Post Serializer to create a new post in the vchat project, removed the id from the fields

```
class PostSerializer(serializers.ModelSerializer):
    comments = CommentSerializer(many=True, read_only=True)
    user = serializers.CharField(source='user.username', read_only=True)
    class Meta:
        model = models.Post
        fields = ['user','post_source', 'caption', 'comments',
'uploaded_at', '>
        read_only_fields = ['uploaded_at', 'likes_count']
```

- Deleting previous users from the DB to verify that everything works fine.
- A little touch up to topic #WebRTC source WebRTC for the Curious.
- Forgot to add this : refactoring the views.py for PostCreateAPIView

```
class PostCreateAPIView(CreateAPIView):
    queryset = models.Post.objects.all()
    serializer_class = serializers.PostSerializer
    permission_classes = [AllowAny]

    def perform_create(self, serializer):
        serializer.save(user=self.request.user)
```

reason for that change is it throws an error when I tried to post a new data to the post Model.