Friyay3

Warm Up Problem

Work individually and come back at 9:15 for us to go over it. You can use Visual Studio Code (not whiteboard practice).

Write a function called **count** that accepts a string and returns a dictionary containing the counts of each character in the string. count('abccc') should return {"a": 1, "b": 1, "c": 3}.

count.py > ...

```python
def count(str):
    countDic = {}
    for character in str:
        if countDic.get(character, False):
            countDic[character] += 1
        else:
            countDic[character] = 1
    return countDic

print(count("abccc")) # {"a": 1, "b": 1, "c": 3}
```

Write a function called **anagram** that accepts two strings and return True if they are anagrams and False if they aren't. anagram('listen', 'silent') should return True. anagram('cat', 'axe') should return False.

Part 1: Solution

```python
def anagram(str1, str2):
    # if lengths aren't same, not anagrams
    if len(str1) != len(str2):
        return False

    # get counts for characters in each string
    strCount1 = {}
    strCount2 = {}

    # since strings are same length, doesn't matter which str I use here
    for index in range(len(str1)):
        character1 = str1[index]
        character2 = str2[index]

        # if dictionary contains character, add 1 to current count, else set count at 1
        if strCount1.get(character1, False):
            strCount1[character1] += 1
        else:
            strCount1[character1] = 1

```

Part 2: Solution

```
        # have to do it for second string also
        if strCount2.get(character2, False):
            strCount2[character2] += 1
        else:
            strCount2[character2] = 1

    # check if value at key in one dictionary is equal to value in second dictionary (are the character counts the same? if not return false)
    for key in strCount1.keys():
        if strCount1[key] != strCount2[key]:
            return False

    # if all counts are the same, return true. MUST BE OUTSIDE LOOP!
    return True
```

Big O Notation

What is Big O and why do we need it?

Big O is the language we use for talking about how long an algorithm takes to run (or how much space it takes up).

We use this notation because these problems have lots of solutions, and we need a way of knowing if one is better than another. We judge this based on speed (and sometimes space).

Calculating Speed of Algorithms:

https://rithmschool.github.io/function-timer-demo/

Computers all have different speeds and runtimes. So instead, we want to count how good an algorithm is in terms of steps. To show speed, let's look at some examples:

O(1 or constant)

return-second.py > ...

```python
def return_second(lst):
    return lst[1]

print(return_second([1, 2, 3, 4, 5, 6, 7]))
```

O(n or linear)

print_list.py > ...

```python
def print_list(lst):
    for thing in lst:
        print(thing)

print_list([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

O(n^2 or quadratic)

quadratic.py > ...

```python
def print_times_table(size_of_table):
    for num1 in range(1, size_of_table + 1):
        for num2 in range(1, size_of_table + 1):
            product = num1 * num2
            print(f'{num1} times {num2} equals {product}')


print_times_table(3)
'''
1 times 1 equals 1
1 times 2 equals 2
1 times 3 equals 3
2 times 1 equals 2
2 times 2 equals 4
2 times 3 equals 6
3 times 1 equals 3
3 times 2 equals 6
3 times 3 equals 9
'''
```

O(log(n) or logarithmic)

logarithmic.py > ...

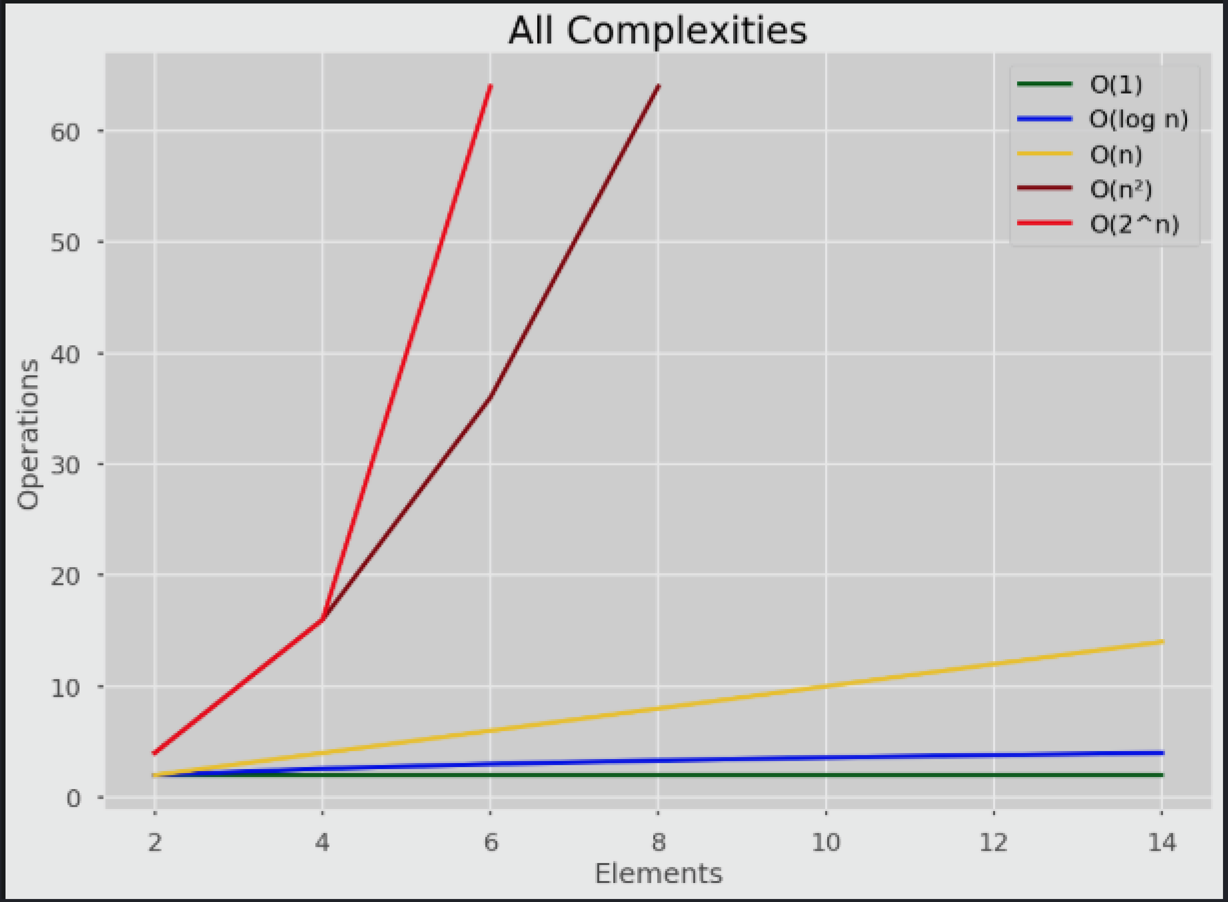
```python
def number_of_halves(num):
    count = 0
    while num > 1:
        num /= 2
        count += 1
    return count

print(number_of_halves(100)) # 100, 50, 25, 12.5, 6.25, 3.125, 1.5625 = 7
```

O(2^(n) or exponential)

It's like trying to guess a password. If a password is 10 characters long, and you have 62 possible characters(A-Z, a-z, 0-9) to choose from, then to find how many tries it would take, it's 62^10 ... aka a really, really big number. Very rarely will you encounter an exponential BigO function

Comparing Big O:



Extra Resources:

https://skerritt.blog/big-o/

https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7