DSC 40B - Homework

Due:

Write your solutions to the following problems by either typing them up or handwriting them on another piece of paper. Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer. Homeworks are due via Gradescope at 11:59 p.m.

Problem 1.

As a data scientist, it is important to have a sense for which algorithms are feasible for a given problem size. For instance: How big of a data set can a quadratic algorithm crunch in one minute? That's what this problem aims to find out.

Suppose Algorithm A performs n basic operations when given an input of size n, while Algorithm B performs n^2 basic operations and Algorithm C performs n^3 basic operations. Suppose that your computer takes 1 nanosecond to perform a basic operation. What is the largest problem size each can solve in 1 second, 10 minutes, and 1 hour? That is, fill in the following table:

	1 sec.	10 min.	1 hr
Algorithm A	?	?	?
Algorithm B	?	?	?
Algorithm C	?	?	?

Show your work for at least one cell in each row.

Note: 1 nanosecond is a very *optimistic* estimate of the time it takes for a computer to perform a basic operation.[1]

Hint: it's a small thing, but your answers should never be decimals. It isn't possible to perform 3.7 operations, after all. Should you round up or round down?

Problem 2.

For each of the following pieces of code, state the time complexity using Θ notation in as simple of terms as possible. You do not need to show your work (but be careful, because without work we can't give you partial credit!).

a)
```
def f_1(n):
    for i in range(1_000_000, n):
        for j in range(i):
            print(i, j)
```

b)
```
def f_2(n):
    for i in range(n, n**5):
        j = 0
        while j < n:
            print(i, j)
            j += 1
```

c)
```
def f_3(numbers):
    n = len(numbers)
```

[1] See Peter Norvig's essay "Teach Yourself Computer Programming in 10 Years" for a table of timings for various operations. http://norvig.com/21-days.html

```
        for i in range(n):
            for j in range(n):
                print(numbers[i], numbers[j])

        if n % 2 == 0: # if n is even...
            for i in range(n):
                print("Unlucky!")
```

d)
```
def f_4(arr):
    """arr is an array of size n"""
    t = 0
    n = len(arr)
    for i in range(n):
        t += sum(arr)

    for j in range(n):
        print(j//t)
```

Problem 3.

Determine the time complexity of the following piece of code, showing your reasoning and your work.

```
def f(n):
    i = 1
    while i <= n:
        i *= 2
        for j in range(i):
            print(i, j)
```

Hint: you might need to think back to calculus to remember the formula for the sum of a geometric progression... or you can check wikipedia.[2]

Problem 4.

Consider the following code which constructs a numpy array of n random numbers:[3]

```
import numpy as np
results = np.array([])
for i in range(n):
    results = np.append(results, np.random.uniform())
```

Remember that we have to write `results = np.append(results, np.random.uniform())` instead of just `np.append(results, np.random.uniform())` because it turns out that `np.append` returns a *copy* of `results` with the new entry appended to the end.

Note that this code is very similar to how we taught you to run simulations in DSC 10: we first created an empty numpy array, and then ran our simulation in a loop, appending the result of each simulation with `np.append`. When we ran simulations, we often used $n = 100{,}000$ or larger (and they took a while to finish).

a) Guess the time complexity of the above code as a function of n. Don't worry about getting the right answer (we won't grade for correctness). You don't need to explain your answer.

b) Time how long the above code takes when n is: 10,000, 20,000, 40,000, 80,000, 120,000, and 160,000.

[2] `https://en.wikipedia.org/wiki/Geometric_progression`

[3] Note that in practice you wouldn't do this with a loop; you'd write `np.random.uniform(n)` to generate the array in one line of code.

Then make a plot of the times, where the x-axis is n (the input size) and the y-axis is the time taken in seconds.

Remember to provide not only your plot, but to show your work by providing the code that generated it.

Hint: You can do the timing by hand with the %%`time` magic function in a Jupyter notebook, or you can use the `time()` function in the `time` module. For example, to time the function `foo`:

```
import time
start = time.time()
foo()
stop = time.time()
time_taken = stop - start
```

c) Looking at your plot, what do you now think the time complexity is? Why does the code have this time complexity?

Hint: what is the time complexity of `np.append`, and why?

d) It turns out that creating an empty numpy array and appending to it at the end of each iteration is a *terrible* way to do things, and you should *never* write code like this if you can avoid it.[4] Instead, you should create an empty Python list, append to it, then make an array from that list, like so:

Remember to provide not only your plot, but to show your work by providing the code that generated it.

```
lst = []
for i in range(n):
    lst.append(np.random.uniform())
arr = np.array(lst)
```

To check this, repeat part (b), but with this new code. Show your plot. It is OK if your plot is a little odd, but it shouldn't be quadratic! (Check with a tutor if you're concerned).

[4] We taught you the `np.append` way because it was conceptually simpler – we didn't need to introduce you to Python lists. This is one instance in which your professors lie to you in early courses, then correct their lies later on.