DSC 40B - Homework 01

Due: Wednesday, April 9

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.

Roughly how long will it take for a linear time algorithm to run? What about a quadratic time algorithm? Or worse, a cubic? In this problem, we'll estimate these times.

Suppose algorithm A takes n microseconds to run on a problem of size n, while algorithm B takes n^2 microseconds and algorithm C takes n^3 microseconds (recall that a microsecond is one millionth of a second). How long will each algorithm take to run when the input is of size one thousand, ten thousand, one hundred thousand, and one million? That is, fill in the following table:

	$n = 1{,}000$	$n = 10{,}000$	$n = 100{,}000$	$n = 1{,}000{,}000$
A (Linear)	0.00 s	0.01 s	0.10 s	1 s
B (Quadratic)	?	?	?	?
C (Cubic)	?	?	?	?

The answers for Algorithm A are already provided; you can use them to check your strategy.

Express each time in either seconds, minutes, hours, days, or years. Use the largest unit that you can without getting an answer less than one. For example, instead of "365 days", say "1 year"; but use "364 days" instead of "0.997 years".

Example: If your answer is "1,057,536 seconds", you would report your answer as "12.24 days". If your answer is "438 seconds", you would report it as "7.3 minutes".

Round to two decimal places (it's OK for an answer to round to 0.00).

Hint: you can calculate your answers by hand, or you can write some code to compute them. If you write code, provide it with your solution – if you solve by hand, show your calculations.

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 doing so might help you earn partial credit in case your overall answer is not correct).

a)
```
def f_1(n):
    for i in range(2025, n):
        for j in range(i):
            print(i, j)
```

b)
```
def f_2(n):
    for i in range(n, n**14):
        j = 0
        while j < n:
            print(i, j)
            j += 1
```

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

e)
```python
def f_5(n):
    ranges = [10, n//2, n, n**2]
    for rng in ranges:
        for i in range(rng):
            print(i)
```

f)
```python
def f_5(n):
    ranges = [10, n//2, n, n**2]
    for rng_a in ranges:
        for rng_b in ranges:
            for i in range(rng_a + rng_b):
                print(i)
```

Problem 3.

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

```python
def f(n):
    i = 1
    while i <= n:
        i *= 3
        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.[1]

Problem 4.

Consider the following code which constructs a numpy array of n random numbers:[2]

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

[2] 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.

```
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. 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.[3] 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).

Problem 5.

Consider the code below:

[3] 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.

```
def foo(n):
    i = 1
    while i**2 < n:
        i += 1
    return i
```

a) What does `foo(n)` compute, roughly speaking?

b) What is the asymptotic time complexity of `foo`?

You do not need to show your work for this problem, but providing an explanation might help earn partial credit in case your answer is incorrect.