DSC 40B - Homework 01

Due: Monday, October 3

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". 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.

Solution:

	$n = 1{,}000$	$n = 10{,}000$	$n = 100{,}000$	$n = 1{,}000{,}000$
A (Linear)	0.00 s	0.01 s	0.10 s	1 s
B (Quadratic)	1 s	1.67 mins	2.78 hrs	11.57 days
C (Cubic)	16.67 mins	11.57 days	31.71 years	3170.98 decades

```python
#Time taken for exactly one run
t = 1/1000000
n1, n2, n3, n4 = 1_000, 10_000, 100_000, 1_000_000
#For Quadratic time
b1 = t*n1**2
b2 = round((t*n2**2)/60, 2) #Convert to minutes
b3 = round((t*n3**2)/3600, 2) #Convert to hours
b4 = round((t*n4**2)/86400, 2) #Convert to Days

#Cubic tme
c1 = round((t*n1**3)/60, 2) #Convert to minutes
c2 = round((t*n2**3)/86400, 2) #Convert to days
c3 = round((t*n3**3)/(86400*365), 2)#Convert to years
c4 = round((t*n4**3)/(86400*365*10), 2)#Convert to decades
```

```
print("For Quadratic = " + str([b1, b2, b3, b4]))
print("For Cubic = " + str([c1, c2, c3, c4]))
```

Problem 2.

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.[1]

Solution: As i increases at a rate of 2^n on the flip side the while loop then runs a total of $\log_2(n)$ times which grants us a time complexity of $\Theta(\log n)$. As for the second loop, i increases at a rate of 2^n however since the outer loop is a while loop that does not allow i to be any more than n, the inner loops time complexity is $\Theta(n)$. Combining the 2 largest time complexities we get the final total time complexity of the algorithm to be $n \times \log n = \Theta(n \log n)$

Problem 3.

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

```
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.

Solution: Looking at the code, the for loop runs a total of n times giving a time complexity of $Theta(n)$ and the inner code copies a numpy array and adds a new value to it, I assume runs in a time complexity of $\Theta(1)$ Therefore we get the final time complexity of the algothrim to be $\Theta(n)$

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.

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`:

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

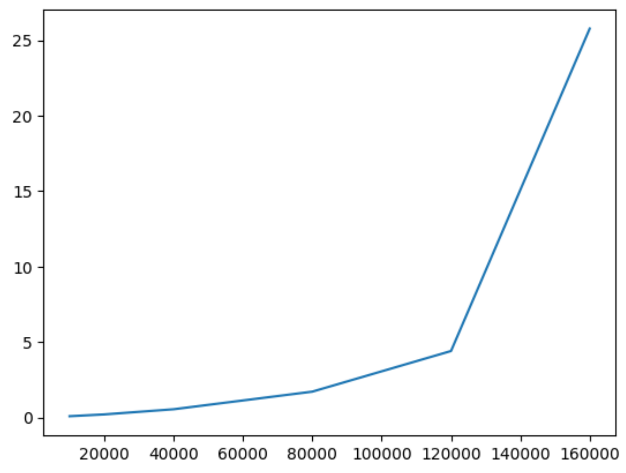
Solution:

```
import numpy as np
import time
import matplotlib.pyplot as plt

#Function
def ranint(n):
    results = np.array([])
    for i in range(n):
        results = np.append(results, np.random.uniform())
    return results



#Loops
iterations = [10_000, 20_000, 40_000, 80_000, 120_000, 160_000]
times = []
for n in iterations:
    start = time.time()
    ranint(n)
    end = time.time()
    times.append(end - start)

#Plot Graph
plt.plot(iterations, times)
```

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?

Solution:



Looking at the plot it is easy to see that the graph shows and exponential increase as the values of n increase. This tells me that the function np.append is not $\Theta(1)$ as I thought at the beginning rather it works in $\Theta(n)$ time, meaning every time it copies an array it loops through each element in the array copying it to a new array. Which then gives us the time complexity of the entire algorithm to be $\Theta(n \times n) = \Theta(n^2)$

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:

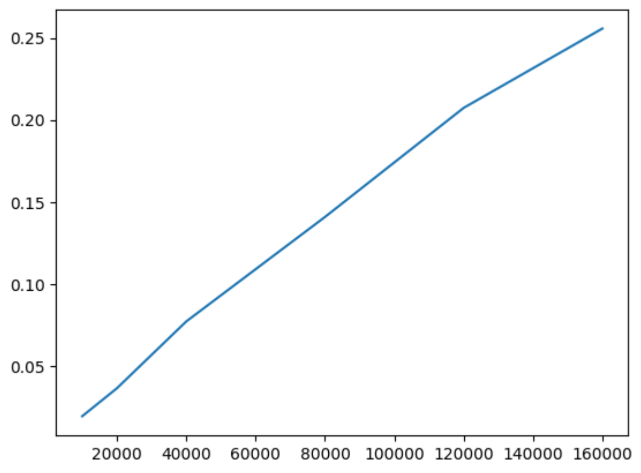
```
lst = []
for i in range(n):
    lst.append(np.random.uniform())
arr = np.array(lst)
```

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).

Solution: Code

```
times = []
for n in iterations:
    start = time.time()
    lst = []
    for i in range(n):
        lst.append(np.random.uniform())
    arr = np.array(lst)
    end = time.time()
    times.append(end - start)

plt.plot(iterations, times)
```

[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.



With the new method, we can see that the time complexity of the same algorithm has went from a quadratic time to a linear time $\Theta(n)$