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?

Solution: First, we convert the times into nanoseconds. One second is 10^9 nanoseconds; ten minutes is $6 \cdot 10^{11}$ nanoseconds; and one hour is $3.6 \cdot 10^{12}$ nanoseconds.

Algorithm A can solve a problem of size n in n nanoseconds. Hence it can solve a problem of size 10^9 in one seconds; a problem of size $6 \cdot 10^{11}$ in ten minutes; and a problem of size $3.6 \cdot 10^{12}$ in one hour.

It takes Algorithm B n^2 nanoseconds to solve a problem of size n. So if we have x nanoseconds of computing time available, the largest problem size we can solve is $\sqrt{x}$. To be more precise, $\sqrt{x}$ might not be an integer, so we should round *down* to get a valid problem size. Rounding down to the nearest integer is also called *taking the floor*, and is written $\lfloor \cdot \rfloor$. So we find that Algorithm B can solve a problem of size $\lfloor \sqrt{10^9} \rfloor = 31{,}622$ in one second; a problem of size $\lfloor \sqrt{6 \cdot 10^{11}} \rfloor = 774{,}596$ in ten minutes; and a problem of size $\lfloor \sqrt{3.6 \cdot 10^{12}} \rfloor = 1{,}897{,}366$ in one hour.

Algorithm C can solve a problem of size n in n^3 nanoseconds. That is, given x nanoseconds, it can solve a problem of size $\lfloor (x)^{1/3} \rfloor$. Therefore, it can solve a problem of size $\lfloor (10^9)^{1/3} \rfloor = 1{,}000$ in one second; a problem of size $\lfloor (6 \cdot 10^{11})^{1/3} \rfloor = 8{,}434.$ in ten minutes; and a problem of size $\lfloor (3.6 \cdot 10^{12})^{1/3} \rfloor = 15{,}326$ in one hour.

Therefore the filled-in table is:

[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`

	1 sec.	10 min.	1 hr.
Algorithm A	10^9	$6 \cdot 10^{11}$	$3.6 \cdot 10^{12}$
Algorithm B	31,622	774,596	1,897,366
Algorithm C	1,000	8,434	15,326

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)
```

Solution: $\Theta(n^2)$.

This is very similar to the dependent loops we saw in the tallest doctor problem where we considered unordered pairs.

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

Solution: $\Theta(n^6)$.

Although we have a `while`-loop, it essentially behaves like a nested `for`-loop over `range(n)`. So we have independent loops, and the total number of executions of the innermost loop body is $\Theta(n^6)$.

c)
```
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!")
```

Solution: $\Theta(n^2)$.

When analyzing the time complexity of a piece of code where each line takes constant time to execute once, we just need to find the line that executes the most number of times. In this case, it's the inner body of the nested `for`-loop. It executes n^2 times, for a time complexity of $\Theta(n^2)$.

We don't know if the last `for` loop will run or not – that depends on whether n is even or odd. But even if it's even (no pun intended) and it *does* run, it won't change the overall time complexity because $n^2 + n$ is still $\Theta(n^2)$.

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)
```

> **Solution:** $\Theta(n^2)$.
>
> This is an example of a piece of code where not every line takes constant time to execute! In particular, the `sum{arr}` line takes $\Theta(n)$ time to execute once, and it is executed n times, for a total time of $\Theta(n^2)$.

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]

> **Solution:** $\Theta(n)$.
>
> How many times does the `print` statement execute? That will tell us the time complexity. On the first iteration of the outer loop, `i = 2` and this line runs twice. On the second iteration of the outer loop, `i = 4` and this line runs four times. On the third iteration, it runs 8 times. In general, on the kth iteration of the outer loop, the line runs 2^k times.
>
> The outer loop will run *roughly* $\log_2 n$ times, since i is doubling on each iteration. We'll see in a moment why it's OK to have a rough estimate instead of being exact here.
>
> The total number of executions of the `print` is therefore:
>
> $$\underbrace{2}_{\text{1st outer iter.}} + \underbrace{4}_{\text{2nd outer iter.}} + \underbrace{8}_{\text{3rd outer iter.}} + \ldots + \underbrace{2^k}_{k\text{th outer iter.}} + \ldots + \underbrace{2^{\log_2 n}}_{\log_2 n\text{th outer iter.}}$$
>
> Or, written using summation notation:
>
> $$\sum_{k=1}^{\log_2 n} 2^k$$
>
> This is the sum of a *geometric progression*. Wikipedia tells us that the formula for the sum of $1 + 2 +$

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

$4 + 8 + \ldots + 2^K$ is:

$$\sum_{k=0}^{K} 2^k = \frac{1 - 2^{K+1}}{1 - 2} = 2^{K+1} - 1$$

Notice that this sum starts from $k = 0$, while ours starts with $k = 1$. In other words, our sum is the same except it is missing the first term corresponding to $k = 0$. So to "correct" this, we subtract the $k = 0$ term (which is $2^0 = 1$) from the larger sum :

$$\sum_{k=1}^{K} 2^k = \left(\sum_{k=0}^{K} 2^k\right) - 2^0 = \left(2^{K+1} - 1\right) - 1 = 2^{K+1} - 2$$

Plugging in $K = \log_2 n$:

$$\sum_{k=1}^{\log_2 n} 2^k = 2^{\log_2 n + 1} - 2$$

Using the fact that $a^{b+c} = a^b \cdot a^c$:

$$\begin{aligned} &= 2^{\log_2 n} \cdot 2 - 2 \\ &= 2n - 2 \\ &= \Theta(n) \end{aligned}$$

Now, remember that we said that we could afford to be a little imprecise when calculating the number of times that the outer loop runs. Let's see why. If n isn't a power of 2, $\log_2 n$ will not be an integer. For instance, if $n = 17$, $\log_2 n = 4.08$. Of course, the loop can't iterate 4.08 times – you can check that it will actually iterate 5 times. In general, the loop will run exactly $\lceil \log_2 n \rceil$ times, where $\lceil \cdot \rceil$ is the *ceiling* operation; it rounds a real number up to the next integer.

In other words, $\log_2 n$ could actually be as much as one less than the actual number of iterations. Perhaps to be careful we should overestimate the number of iterations to be $\log_2 n + 1$. What if we were to use $\log_2 n + 1$ instead? We'd end up with:

$$\sum_{k=1}^{\log_2 n + 1} 2^k = 2^{\log_2 n + 2} - 2 = 4 \cdot 2^{\log_2 n} - 2 = \Theta(n) = 4n - 2$$

So the time complexity doesn't change. This is why using Θ notation allows us to be "sloppy" at times without being incorrect. It saves us work, as long as we know how to use it correctly!

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.

[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.

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: I don't know... $\Theta(n)$?

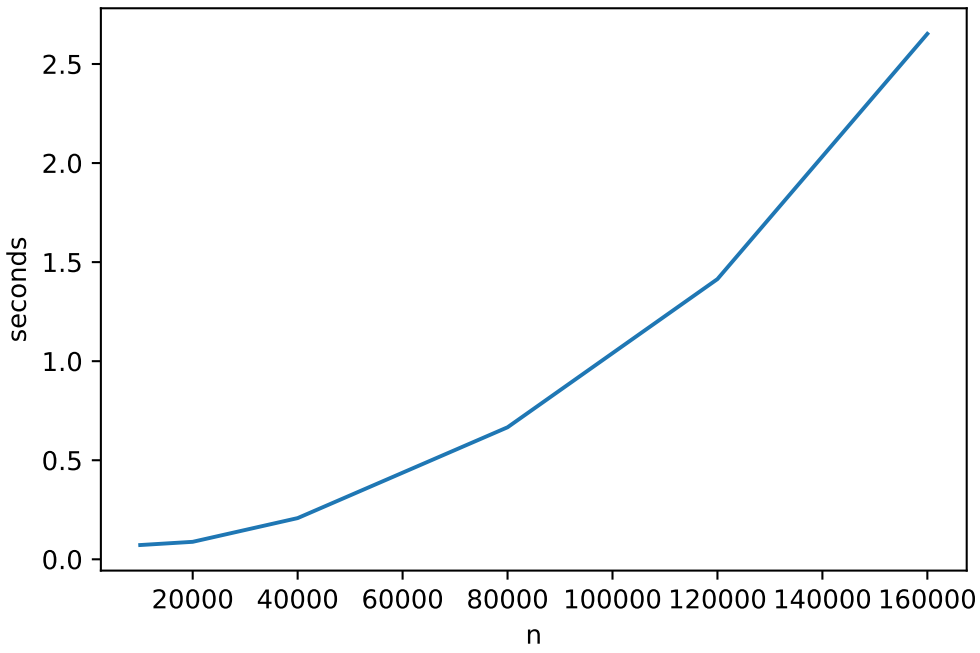
b) Time how long the above code takes when n is: 10,000, 20,000, 40,000, 80,000, 120,000, and 160,000. Then make a plot of the times, where the x-axis is n (the input size) and the y-axis is the time taken in seconds.

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
```

Solution: You should get a plot that looks like this:



This plot was generated by the following code:

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

```python
def foo(n):
    arr = np.array([])
    for i in range(n):
        arr = np.append(arr, np.random.uniform())
    return arr

def time_taken(n):
    start = time.time()
    foo(n)
    stop = time.time()
    return stop - start

sizes = [10_000, 20_000, 40_000, 80_000, 120_000, 160_000]
times = [time_taken(n) for n in sizes]

plt.plot(sizes, times)
plt.xlabel("n")
plt.ylabel("seconds")
plt.savefig("times.pdf")
```

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: It takes $\Theta(n^2)$ time (quadratic).

This is because `np.append` takes linear time since it makes a copy of the array. On the first iteration of the loop, the array is empty and copying it takes no time. One the second iteration, one element has to be copied. On the third iteration, two elements have to be copied... and so on. On the nth iteration, $n-1$ elements have to be copied – which might take some time!

This is therefore an example of a dependant nested loop where the inner loop body takes $\Theta(i)$ time. This gives an arithmetic sum for the time complexity

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:

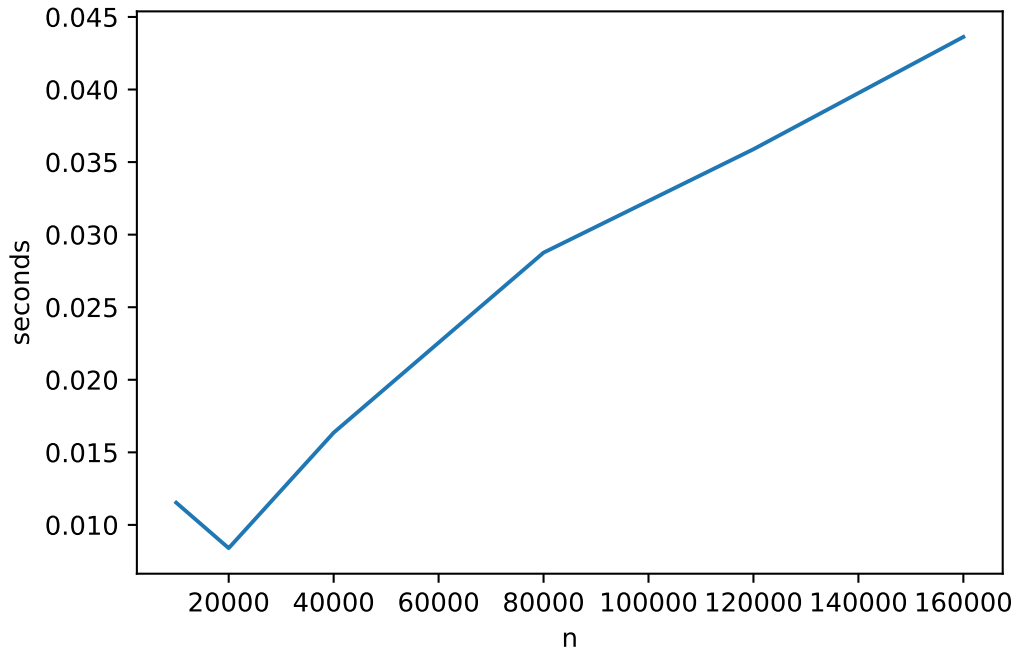
Remember to provide not only your plot, but to show your work by providing the code that generated it.

```python
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: You should get something like this:

[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.



It turns out that this approach has linear time complexity (to be specific, it has *amortized* linear time complexity).