DSC 40B - Homework 01

Due: Wednesday, October 8

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.

Solution:

	$n = 1{,}000$	$n = 10{,}000$	$n = 100{,}000$	$n = 1{,}000{,}000$
A	0.00 s	0.01 s	0.10 s	1 s
B	1 s	1.67 m	2.78 h	11.57 d
C	16.67 m	11.57 d	31.71 y	31,709.79 y

This was computed by the following code:

```
sizes = [1_000, 10_000, 100_000, 1_000_000]

MS_IN_S = 1_000_000
MS_IN_M = MS_IN_S * 60
MS_IN_H = MS_IN_M * 60
MS_IN_D = MS_IN_H * 24
MS_IN_Y = MS_IN_D * 365

MS_IN = {
```

```
    'years': MS_IN_Y,
    'days': MS_IN_D,
    'hours': MS_IN_H,
    'minutes': MS_IN_M,
    'seconds': MS_IN_S,
}

def time_taken(f):
    return [humanize(f(s)) for s in sizes]

def humanize(ms):
    for unit, amount in MS_IN.items():
        if ms / amount >= 1:
            return f'{ms / amount:0.2f} {unit}'
    return f'{ms / amount:0.2f} {unit}'

# linear time
time_taken(lambda: n)

# quadratic
time_taken(lambda: n**2)

# cubic
time_taken(lambda: n**3)
```

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)
```

> **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**14):
        j = 0
        while j < n:
            print(i, j)
            j += 1
```

> **Solution:** $\Theta(n^15)$.
>
> 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 $\Theta(n^15)$.

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 $\Theta(n^2)$.

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

Solution: This outer `for`-loop makes 4 iterations – one for each element of `ranges`.

On the first iteration of the outer loop, the inner loop makes 10 iterations, taking a total of 10 units of time. That is, the first iteration takes constant time.

On the second iteration of the outer loop, the inner loop makes roughly $n/2$ iterations, taking a total of $\Theta(n/2) = \Theta(n)$ time.

On the third iteration of the outer loop, the inner loop makes n iterations, taking $\Theta(n)$ time.

On the fourth iteration of the outer loop, the inner loop makes n^2 iterations, taking $\Theta(n^2)$ time.

In total, the time taken is $\Theta(1) + \Theta(n) + \Theta(n) + \Theta(n^2) = \Theta(n^2)$.

f)
```
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)
```

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

The innermost loop, `for i in range(rng_a + rng_b)`, is executed 16 times (once for each ordered pair of ranges from `ranges`), making a different number of executions each time.

For example, one execution of the innermost loop will be when `rng_a = n` and `rng_b = n**2`. In this case, the innermost loop will make $n + n^2$ iterations, taking $\Theta(n^2)$ time.

From there, you might see that the overall time complexity will be $\Theta(n^2)$, because of the 16 executions of the innermost loop, the most costly will be the one where both `rng_a` and `rng_b` are n^2. In this situation, the innermost loop will make $n^2 + n^2 = 2n^2$ iterations, taking $\Theta(n^2)$ time. All of the other executions are $\Theta(n^2)$ or smaller, so the total time taken will be $\Theta(n^2)$.

Alternatively, you could write out the time taken by each of the 16 executions of the innermost loop. Those 16 executions are:

`rng_a`	`rng_b`	# iterations	time taken
10	10	20	$\Theta(1)$
10	$n/2$	$10 + n/2$	$\Theta(n)$
10	n	$10 + n$	$\Theta(n)$
10	n^2	$10 + n^2$	$\Theta(n^2)$
$n/2$	10	$n/2 + 10$	$\Theta(n)$
$n/2$	$n/2$	$n/2 + n/2$	$\Theta(n)$
$n/2$	n	$n/2 + n$	$\Theta(n)$
$n/2$	n^2	$n/2 + n^2$	$\Theta(n^2)$
n	10	$n + 10$	$\Theta(n)$
n	$n/2$	$n + n/2$	$\Theta(n)$
n	n	$n + n$	$\Theta(n)$
n	n^2	$n + n^2$	$\Theta(n^2)$
n^2	10	$n^2 + 10$	$\Theta(n^2)$
n^2	$n/2$	$n^2 + n/2$	$\Theta(n^2)$
n^2	n	$n^2 + n$	$\Theta(n^2)$
n^2	n^2	$n^2 + n^2$	$\Theta(n^2)$

Adding up the 16 elements in the rightmost column, we get $\Theta(n^2)$.

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 *= 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]

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 = 3` and this line runs three times. On the second iteration of the outer loop, `i = 9` and this line runs nine times. On the third iteration, it runs 27 times. In general, on the kth iteration of the outer loop, the line runs 3^k times.

The outer loop will run *roughly* $\log_3 n$ times, since i is tripled 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:

$$\underbrace{3}_{\text{1st outer iter.}} + \underbrace{9}_{\text{2nd outer iter.}} + \underbrace{27}_{\text{3rd outer iter.}} + \ldots + \underbrace{3^k}_{k\text{th outer iter.}} + \ldots + \underbrace{3^{\log_3 n}}_{\log_3 n\text{th outer iter.}}$$

Or, written using summation notation:

$$\sum_{k=1}^{\log_3 n} 3^k$$

This is the sum of a *geometric progression*. Wikipedia tells us that the formula for the sum of $1 + 3 + 9 + 27 + \ldots + 3^K$ is:

$$\sum_{k=0}^{K} 3^k = \frac{1 - 3^{K+1}}{1 - 3} = \frac{3^{K+1} - 1}{2} = \frac{3^{K+1}}{2} - \frac{1}{2}$$

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 $3^0 = 1$) from the larger sum :

$$\sum_{k=1}^{K} 3^k = \left(\sum_{k=0}^{K} 3^k\right) - 3^0 = \left(\frac{3^{K+1}}{2} - \frac{1}{2}\right) - 1 = \frac{3^{K+1}}{2} - \frac{3}{2}$$

Plugging in $K = \log_3 n$:

$$\sum_{k=1}^{\log_3 n} 3^k = \frac{3^{\log_3 n + 1}}{2} - \frac{3}{2}$$

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

$$\begin{aligned} &= 3^{\log_3 n} \cdot \frac{3}{2} - \frac{3}{2} \\ &= \frac{3}{2} n - \frac{3}{2} \\ &= \Theta(n) \end{aligned}$$

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 3, $\log_3 n$ will not be an integer. For instance, if $n = 17$, $\log_3 n = 2.58$. Of course, the loop can't iterate 2.58 times – you can check that it will actually iterate 3 times. In general, the loop will run exactly $\lceil \log_3 n \rceil$ times, where $\lceil \cdot \rceil$ is the *ceiling* operation; it rounds a real number up to the next integer.

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

In other words, $\log_3 n$ 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 $\log_3 n + 1$. What if we were to use $\log_3 n + 1$ instead? We'd end up with:

$$\sum_{k=1}^{\log_3 n+1} 3^k = \frac{3^{\log_3 n+2}}{2} - \frac{3}{2} = \frac{9}{2}n - \frac{3}{2} = \Theta(n)$$

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:[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: 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.

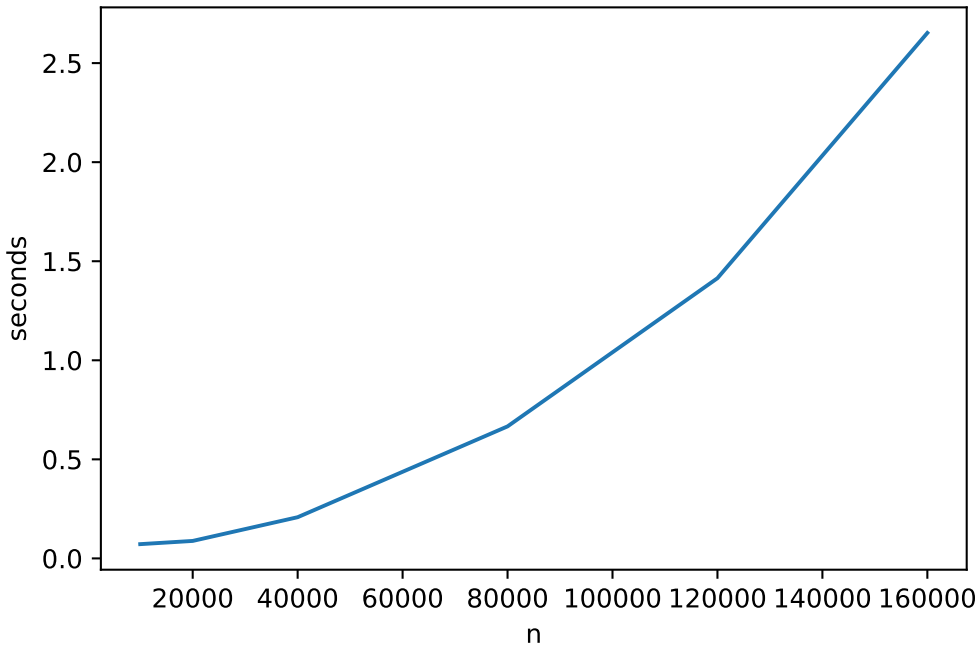
Remember to provide not only your plot, but to show your work by providing the code that generated it.

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:

[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.



This plot was generated by the following code:

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

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

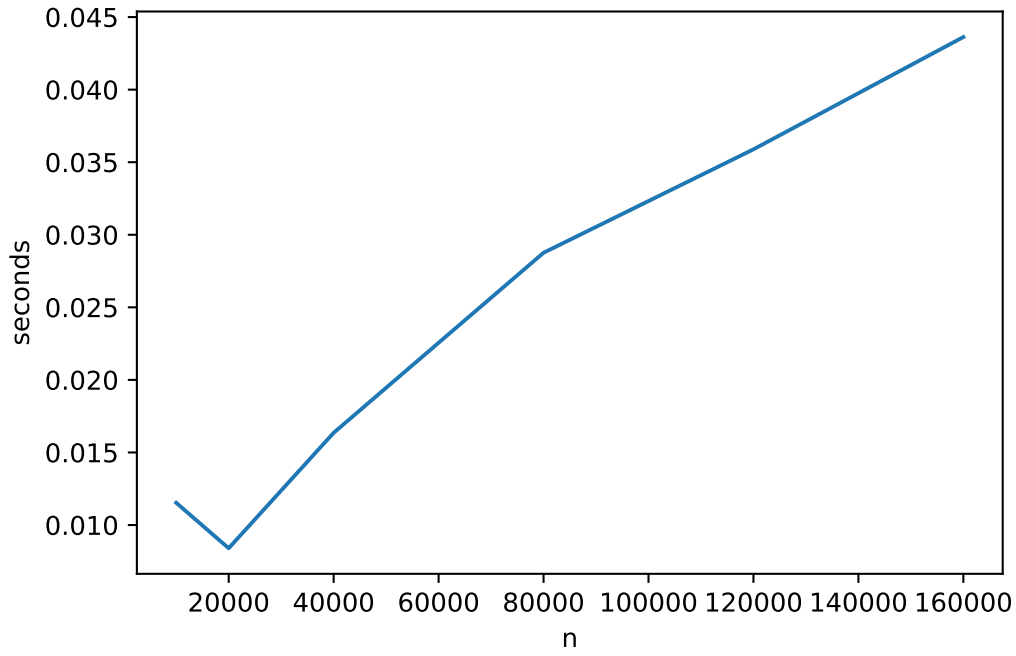
d) It turns out that creating an empty numpy array and appending to it at the end of each iteration is a *terrible* way to do things, and you should *never* write code like this if you can avoid it.[3] Instead, you should create an empty Python list, append to it, then make an array from that list, like 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).

Solution: You should get something like this:



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

[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.

Problem 5.

Consider the code below:

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

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

> **Solution:** It computes, approximately, $n^{1/2}$. Of course, `foo` always returns an integer, so the result of the function is usually not exactly correct.
>
> More precisely, `foo` returns the smallest integer greater than or equal to the square root $n^{1/2}$. For example, $(10)^{1/2}$ is between 3 and 4; `foo(10)` returns 4.

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.

> **Solution:** $\Theta(n^{1/2})$