DSC 40B - Homework 01

Due: Wednesday, January 18

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	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.

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: $\Theta(n)$.

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:

$$\underbrace{2}_{\text{1st outer iter.}} + \underbrace{4}_{\text{2nd outer iter.}} + \underbrace{8}_{\text{3rd outer iter.}} + \ldots + \underbrace{2^k}_{k\text{th outer iter.}} + \ldots + \underbrace{2^{\log_2 n}}_{\log_2 n\text{th outer iter.}}$$

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

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 $1 + 2 + 4 + 8 + \ldots + 2^K$ 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 3.

Consider the code below:

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

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

Solution: It computes, approximately, $\sqrt{n}$. Of course, `foo` always returns an integer, so the result of the function is usually not exactly correct.

More precisely, `foo` returns the largest integer greater than or equal to $\sqrt{n}$. For example, $\sqrt{5}$ is between 2 and 3; `foo(5)` returns 3.

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

Solution: $\Theta(\sqrt{n})$