DSC 40B - Homework 01

Due: Wednesday, April 10

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.

Suppose

- algorithm A takes n milliseconds to run on a problem of size n,
- algorithm B takes n^2 milliseconds to run on a problem of size n,
- algorithm C takes 2^n milliseconds to run on a problem of size n,
- algorithm D takes $\log_2(n)$ milliseconds to run on a problem of size n.

What is the largest problem size each algorithm can solve in 1 second, 10 seconds, and 1 minute? That is, fill in the following table:

	1 sec.	10 sec.	1 min
A	?	?	?
B	?	?	?
C	?	?	?
D	?	?	?

Solution: First, we convert the times into milliseconds. One second is 1,000 milliseconds; ten seconds is 10,000 milliseconds, and one minute is 60,000 milliseconds.

Algorithm A can solve a problem of size n in n milliseconds. Hence It can solve a problem of size $1,000$ in $1,000$ milliseconds; a problem of size $10,000$ in $10,000$ milliseconds, and a problem of size $60,000$ in $60,000$ milliseconds.

Algorithm B can solve a problem of size n in n^2 milliseconds. Equivalently, it can solve a problem of size $\sqrt{n}$ in n milliseconds. Therefore, it can solve a problem of size $\sqrt{1,000}$ in 1,000 milliseconds. But $\sqrt{1,000}$ isn't an integer, and problem sizes have to be integers. Hence we take the largest integer smaller than $\sqrt{1,000}$ (also known as the *floor*), which turns out to be 31. The algorithm can solve a problem of size $\sqrt{10,000} = 100$ in 10,000 milliseconds, and a problem of size $\lfloor\sqrt{60,000}\rfloor = 244$ in 60,000 milliseconds.

Algorithm C can solve a problem of size n in 2^n milliseconds. Equivalently, it can solve a problem of size $\log_2(n)$ in n milliseconds. Therefore, it can solve a problem of size $\lfloor\log_2(1,000)\rfloor = 9$ in 1,000 milliseconds; a problem of size $\lfloor\log_2(10,000)\rfloor = 13$ in 10,000 milliseconds; and a problem of size $\lfloor\log_2(60,000)\rfloor = 15$ in 60,000 milliseconds.

Algorithm D can solve a problem of size n in $\log_2(n)$ milliseconds. Equivalently, it can solve a problem of size 2^n in n milliseconds. Therefore, it can solve a problem size of $2^{1,000}$ in 1,000 milliseconds; a problem of size $2^{10,000}$; in 10,000 milliseconds; a problem of size $2^{60,000}$ in 60,000 milliseconds.

Therefore the filled-in table is:

	1 sec.	10 sec.	1 min
A	1,000	10,000	60,000
B	31	100	244
C	9	13	15
D	$2^{1,000}$	$2^{10,000}$	$2^{60,000}$

Problem 2.

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

```python
def f(n):
    i = 1
    while i <= n:
        i *= 2
        for j in range(i**2): # <-- note the range!
            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: $\Theta(n^2)$.

How many times does the `print` statement execute? That will tell us the time complexity, because each line of code takes constant time to run once, and the `print` statement is the line that is run most.

On the first iteration of the outer loop, `i = 2` and this line runs $i^2 = 2^2 = 4$ times. On the second iteration of the outer loop, `i = 4` and this line runs $4^2 = 16$ times. On the third iteration, it runs 64 times. The pattern is captured by the table below:

Iteration #	i	# of `prints`
1	2	4
2	4	16
3	8	64
4	16	256
⋮	⋮	⋮
k	2^k	$2^{2k} = 4^k$
⋮	⋮	⋮

The third column tells us the number of times `print` is executed *during* that iteration of the `while` loop. In general, on the kth iteration of the outer loop, the `print` runs 2^{2k} times, which happens to equal 4^k. The total number of `prints` will be the sum of the third column.

We need to sum up the third column, but we don't yet know the last term in that sum. To figure it out, recognize that 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:

$$\underbrace{4}_{\text{1st outer iter.}} + \underbrace{16}_{\text{2nd outer iter.}} + \underbrace{64}_{\text{3rd outer iter.}} + \ldots + \underbrace{4^k}_{k\text{th outer iter.}} + \ldots + \underbrace{4^{\log_2 n}}_{\log_2 n\text{th outer iter.}}$$

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

Or, written using summation notation:

$$\sum_{k=1}^{\log_2 n} 4^k$$

This is the sum of a *geometric progression.* Wikipedia tells us that the formula for the sum of $1 + 4 + 8 + 16 + \ldots + 4^K$ is:

$$\sum_{k=0}^{K} 4^k = \frac{1 - 4^{K+1}}{1 - 4} = \frac{4^{K+1} - 1}{3}$$

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 $4^0 = 1$) from the larger sum :

$$\sum_{k=1}^{K} 4^k = \left(\sum_{k=0}^{K} 4^k\right) - 4^0 = \frac{4^{K+1} - 1}{3} - 1 = \frac{4^{K+1} - 4}{3}$$

Plugging in $K = \log_2 n$:

$$\sum_{k=1}^{\log_2 n} 4^k = \frac{4^{\log_2 n+1} - 4}{3}$$

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

$$\begin{aligned}
&= \frac{4 \cdot 4^{\log_2 n} - 4}{3} \\
&= \frac{4 \cdot 4^{\log_2 n}}{3} - \frac{4}{3} \\
&= \Theta(4^{\log_2 n})
\end{aligned}$$

Simplifying using the fact that $(a^b)^c = a^{b \cdot c}$:

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

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: an extra term in the sum, which we can split off:

$$\sum_{k=1}^{\log_2 n+1} 4^k = \left(\sum_{k=1}^{\log_2 n} 4^k\right) + 4^{\log_2 n+1}$$

The part in parentheses is the sum we calculated above, which we know is $\Theta(n^2)$. The "extra" term, $4^{\log_2 n+1}$, is equal to $4 \cdot 4^{\log_2 n}$, which is $4n^2$. So the total is still

$$\Theta(n^2) + 4n^2 = \Theta(n^2)$$

and 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**3 < n - 100:
        i += 1
    return i
```

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

Solution: It computes, approximately, $(n - 100)^{1/3}$. 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 the cubic root $(n-100)^{1/3}$. For example, $(110 - 100)^{1/3} = 10^{1/3}$ is between 2 and 3; `foo(110)` returns 3.

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

Solution: $\Theta(n^{1/3})$