DSC 40B - Homework 01

Due: Wednesday, January 12

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.

In lecture, we saw the following algorithm for computing a median:

```
def median(numbers):
    min_h = None
    min_value = float('inf')
    for h in numbers:
        total_abs_loss = 0
        for x in numbers:
            total_abs_loss += abs(x - h)
        if total_abs_loss < min_value:
            min_value = total_abs_loss
            min_h = h
    return min_h
```

We said that this algorithm has quadratic time complexity, which means that the time it takes to run grows like n^2, where n is the size of the input list. Let's see if this is indeed the case by timing the function on inputs of various size.

a) Write a function named `time_median` which takes in a single argument, `n`. It should create a list of `n` numbers and call `median` ten times on that list, timing each call. Your function should return the average time it took for `median` to run over all ten calls. Provide your code.

Hint: We saw the `timeit` magic functions in lecture, but the `time` function in Python's `time` module is more useful here. To time the execution of a function call, we can write:

```
import time
start = time.time()
my_function_that_i_want_to_time()
end = time.time()
elapsed = end - start
```

Solution:

```
def time_median(n):
    # it's also valid to produce any other list with length n
    lst = [random.random() for _ in range(n)]
    elapses = []
    for _ in range(10):
        start = time.time()
        med = median(lst)
        end = time.time()
        elapses.append(end - start)
    return sum(elapses)/10
```

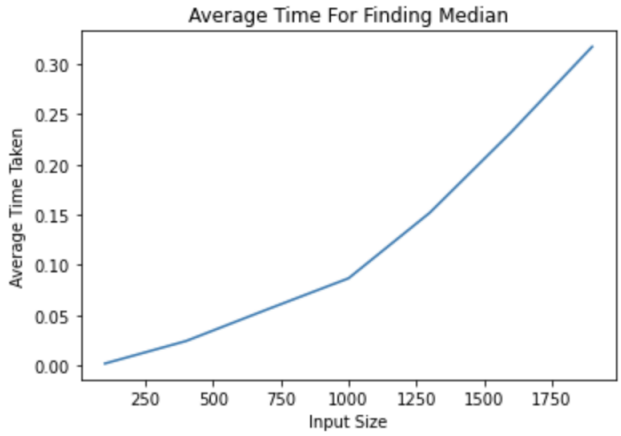
b) Using your function `time_median`, time `median` on inputs of size 100, 400, 700, 1000, 1300, 1600, and 1900. Create a plot of these times, where the horizontal axis measures the input size, and the vertical axis measures the average time taken by `median` in seconds.

Include your code in your solution. You may use any tool you like to make the plot, though we recommend using `matplotlib`.

Solution: Here is some code that will compute the timings.

```
sizes = [100, 400, 700, 1000, 1300, 1600, 1900]
timings = []
for s in sizes:
    timings.append(time_median(s))
plt.plot(sizes, timings)
plt.xlabel('Input Size')
plt.ylabel('Average Time Taken')
plt.title('Average Time For Finding Median')
```

This will produce a figure like that below:



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.

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

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 $1 + 2 + 4 + 8 + \ldots + 2^K$ 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!