CSC110 Tutorial 8: Asymptotic Notation and Algorithm Running-Time Analysis

Print this handout

In this tutorial, you'll review the different concepts you learned about this week: Big-O, Omega, and Theta for comparing the asymptotic (i.e., long-term) growth of functions, and the various techniques we use to analyse the running-time of algorithms. In the last part of the tutorial, you will compare your running-time analyses against empirical timing experiments that you will run using Python's `timeit` module.

Exercise 1: Practice with Asymptotic Notation

1. Consider the following statement:

$$\forall a, b \in \mathbb{R}^+,\ a \geq b \Rightarrow a^n \in \Omega(b^n)$$

   a. Rewrite the above statement, but with the definition of Omega expanded.

   b. Prove the above statement. (Hint: your proof body should actually be quite short, but is good practice with setting up Big-O/Omega/Theta proofs.)

2. Consider the following statement:

$$\forall f, g : \mathbb{N} \to \mathbb{R}^{\geq 0},\ f + g \in \mathcal{O}(\max(f, g))$$

   Here, $f + g$ refers to the function $\mathbb{N} \to \mathbb{R}^{\geq 0}$ defined as $(f+g)(n) = f(n) + g(n)$, and $\max(f, g)$ refers to the function $(\max(f, g))(n) = \max(f(n), g(n))$.

   a. Rewrite the above statement, but with the definition of Big-O expanded.

   b. Prove the above statement.

   Note: it's actually possible to prove that $f + g \in \Theta(\max(f, g))$, although we haven't asked you to prove it here.

Exercise 2: "Debugging" corner and algorithm running-time analysis

Each of the functions below has an *incorrect* running-time analysis written below it. For each of these functions:

- State the error(s) in the running-time analysis.
- Perform a correct running-time analysis for the function.

```
def f1(n: int) -> int:
    """Preconditions: n % 2 == 0 and n >= 2"""
    k = 1

    for i in range(0, n):
        for j in range(n - 2, n):
            k += 1

    return k
```

INCORRECT analysis. There are two basic operations outside the for loop: an assignment statement `k = 1` (1 step) and a return statement `return k` (1 step).

The inner loop has n iterations and 1 step per iteration. So in total the inner loop takes $n * 1 = n$ steps.

The outer loop has n iterations, and each of its iterations take n steps (i.e., the inner loop).

So, $RT_{f1}(n) = 1 + n \cdot n + 1 = n^2 + 2$, which means $RT_{f1} \in \Theta(n^2)$

```
def f2(n: int) -> None:
    """Precondition: n > 1"""
    for i in range(n, 2 * n):
        print(i)

    for j in range(0, n - 1):
        print(j)
```

INCORRECT analysis. The first for loop iterates n times, and each iteration takes 1 step because the loop body is constant time. So the total number of steps for the first for loop is n.

The second for loop will iterate $n - 1$ times, with 1 step per iteration, for a total of $n - 1$ steps.

So, $RT_{f2}(n) = n \cdot (n-1) = n^2 - n$ steps. And so $RT_{f2}(n) \in \Theta(n^2)$.

```
def f3(n: int) -> int:
    """Precondition: n >= 0"""
    sum_so_far = 0

    for i in range(0, n * n):
        if sum_so_far >= 0:
            return sum_so_far
        else:
            sum_so_far += i

    return sum_so_far
```

INCORRECT analysis. The initial assignment statement and final return statement each take constant time (1 step each).

The for loop runs for n^2 iterations. Each iteration takes constant time (1 step), since the loop body consists of only constant-time operations. So the for loop takes n^2 steps.

Then the total number of steps is $1 + n^2 + 1 = n^2 + 2 \in \Theta(n^2)$.

Exercise 3: "Breaking" cryptosystems: efficiency and profiling

In lectures last week, you implemented a brute-force algorithm for breaking the Diffie-Hellman key exchange. Here is one poassible implementation of this algorithm:

```
def break_diffie_hellman(p: int, g: int, g_a: int, g_b: int) -> int:
    """Return the shared Diffie-Hellman secret key obtained from the eavesdropped inform

    Preconditions:
        - p, g, g_a, and g_b are the values exhanged between Alice and Bob
          in the Diffie-Hellman algorithm

    >>> p = 23
    >>> g = 2
    >>> g_a = 9  # g ** 5 % p
    >>> g_b = 8  # g ** 14 % p
    >>> break_diffie_hellman(p, g, g_a, g_b)  # g ** (5 * 14) % p
    16
    """
    secret_a = 1
    for possible_a in range(1, p):
        if pow(g, possible_a, p) == g_a:
            secret_a = possible_a

    # Note: 1 <= secret_a < p
    return pow(g_b, secret_a, p)
```

[Note: Now is an excellent time to review the Diffie-Hellman key exchange before moving on!]

1. Running-time analysis

The above algorithm for `break_diffie_hellman` uses the built-in function `pow`. If we want to analyse the running time of this algorithm, we need to take into account the running time of `pow`.

1. Analyse the running time of `break_diffie_hellman`, *assuming* that `pow` always takes **1 step** (i.e., $\Theta(1)$ time), regardless of its arguments.

   Note that there are four arguments to `break_diffie_hellman`; which argument(s) influence the running time of this function? (Make sure to check your answer with your TA/classmates before moving on.)

2. Now, redo your running-time analysis of `break_diffie_hellman`, assuming that `pow(base, e, n)` takes e **steps** (i.e., $\Theta(e)$ time), regardless of its `base` and `n` arguments.

3. Finally, redo your running-time analysis of `break_diffie_hellman` one more time, assume that `pow(base, e, n)` takes $\log_2 e$ **steps** (i.e., $\Theta(\log e)$ time), regardless of its `base` and `n` arguments.

   You may find the following Theta bound useful: $\sum_{i=1}^{n} \log_2(i) \in \Theta(n \log n)$.

Now, the above running-time analyses are unsatisfying because we don't know which (if any) of our assumptions about the running time of `pow` are valid! It turns out that that the third running time for `pow` ($\Theta(\log e)$) is correct, though in practice the exponent size is typically so small that we can treat `pow` as being constant-time. But, we wanted you to go through the exercise of doing these three analyses to get a sense of how the running-time of a helper function can impact a running-time analysis.

2. Timing experiments

Next, we'll investigate a different way of determining the running-time of an algorithm: performing timing experiments to measure the actual amount of time taken when we call a function.

We can use the Python module `timeit` to measure how long Python code takes to execute on our machine.

```
>>> import timeit
>>> timeit.timeit('5 + 15', number=1000)
1.9799976143985987e-05
```

The function `timeit.timeit` takes a string containing a Python statement (in our example, the simple expression `5 + 15`) and a number of times to execute that statement. When we call the function, it executes the statement the specified number of times, and returns the total time elapsed, in seconds. Of course, as we discussed in Chapter 9, this time measurement is inexact, depending on both the machine you run it on, and what else is currently running on your computer's operating system. (Try it on your own computer to see how you compare!) Even repeating the same `timeit` call results in different values:

```
>>> timeit.timeit('5 + 15', number=1000)
1.4399999999525903e-05
>>> timeit.timeit('5 + 15', number=1000)
1.3100000000321188e-05
>>> timeit.timeit('5 + 15', number=1000)
1.8700000001814487e-05
```

These timing experiments are inexact, but can serve to give us rough estimates of how the performance of our functions grow with the input size. You'll see how in the rest of the tutorial.

To start, please download the starter files `tutorial8.py`, `diffie_hellman_runs.csv`, and `diffie_hellman_runs_large.csv`, and save them into this week's tutorial folder.

1. In `tutorial8.py`, complete the function `time_to_break_diffie_hellman`, which uses `timeit` to measure how long `break_diffie_hellman` takes to run. The doctest shows a sample call (though not the return value—why?).

2. Next, open the file `diffie_hellman_runs.csv`. Each row of this CSV file contains a record of the communication exchanged by one run of Diffie-Hellman (imagine you're the eavesdropper now!!). The order of the numbers matches the parameter order of `break_diffie_hellman`.

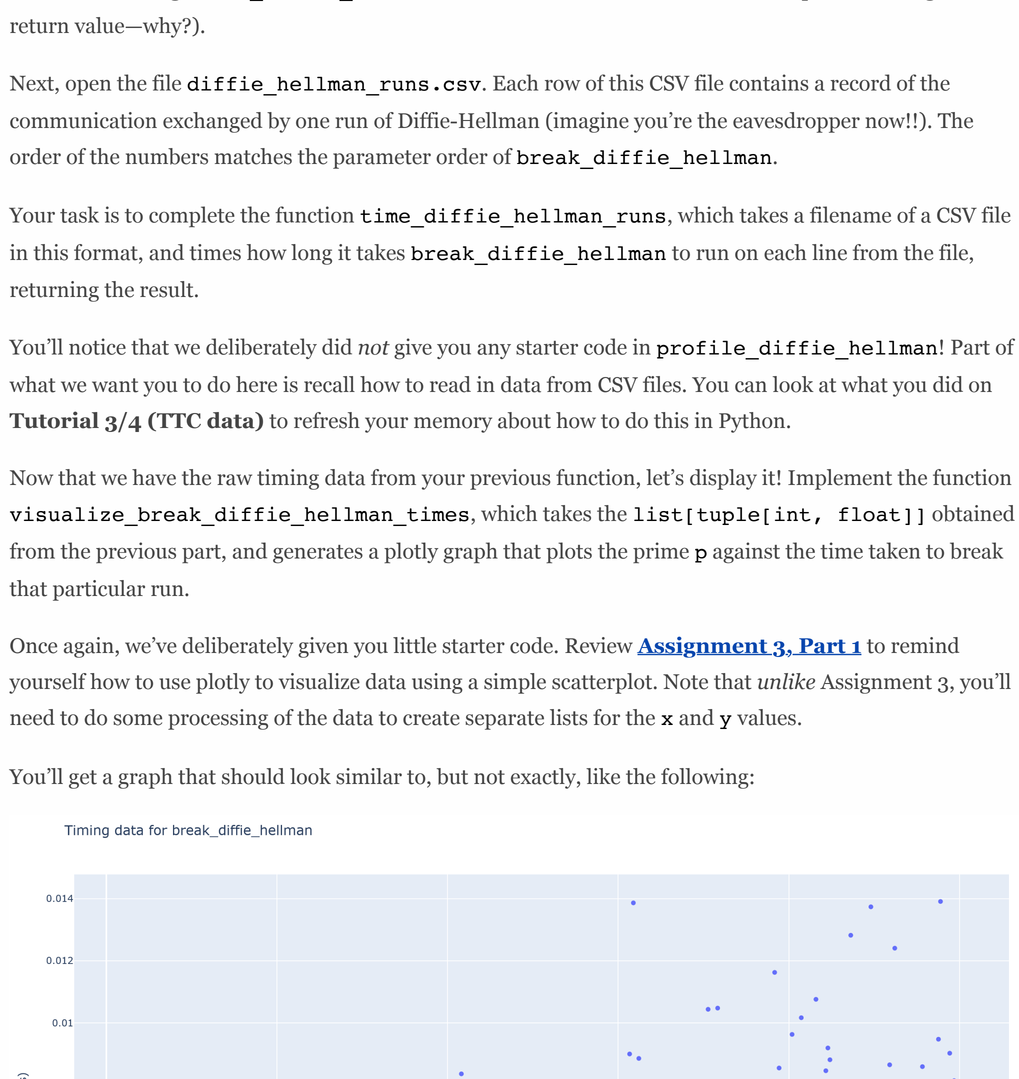
   Your task is to complete the function `time_diffie_hellman_runs`, which takes a filename of a CSV file in this format, and times how long it takes `break_diffie_hellman` to run on each line from the file, returning the result.

   You'll notice that we deliberately did *not* give you any starter code in `profile_diffie_hellman`! Part of what we want you to do here is recall how to read in data from CSV files. You can look at what you did on **Tutorial 3/4 (TTC data)** to refresh your memory about how to do this in Python.

3. Now that we have the raw timing data from your previous function, let's display it! Implement the function `visualize_break_diffie_hellman_times`, which takes the `list[tuple[int, float]]` obtained from the previous part, and generates a plotly graph that plots the prime `p` against the time taken to break that particular run.

   Once again, we've deliberately given you little starter code. Review **Assignment 3, Part 1** to remind yourself how to use plotly to visualize data using a simple scatterplot. Note that *unlike* Assignment 3, you'll need to do some processing of the data to create separate lists for the `x` and `y` values.

   You'll get a graph that should look similar to, but not exactly, like the following:



4. Does the relationship between prime size and time elapsed seem to match the Theta bound you found in your running time analysis?

   What might explain the "outliers" in your scatterplot?

Finally, if you are feeling adventurous and have some time, try using the `diffie_hellman_runs_large.csv` file instead. But be careful: it could take several minutes to run `break_diffie_hellman` on the full file!

Additional exercises

1. Prove that $\forall f, g : \mathbb{N} \to \mathbb{R}^{\geq 0},\ f + g \in \Omega(\max(f, g))$.

2. In our `break_diffie_hellman` implementation, you might notice that once we've found the `secret_a` value in the loop, there's no reason to keep going with any more iterations.

   Try moving the return statement into the if branch in the loop, and re-run your timing experiments. What do you notice?

   We'll discuss in class formally next week how to handle this kind of "early returning" for loop.