9.6 Analyzing Comprehensions and While Loops

In the previous section, we began our study of algorithm running time analysis by looking at functions that are implemented using for loops. We chose for loops as a starting point because they make explicit the *repeated statements* that occur when we execute a function body, while also being relatively straightforward to analyze because of their predicable iteration patterns.

In this section, we'll extend what we learned about for loops to two different kinds of Python code: comprehension expressions and while loop. We'll see how all three obey similar patterns when it comes to repeating code, but while loops offer both more flexibility and more complexity in what they can do.

Comprehensions

Consider the following function:

```
def square_all(numbers: list[int]) -> list[int]:
    """Return a new list containing the squares of the giv
    return [x ** 2 for x in numbers]
```

Running time analysis. How do we analyze the running time of this code? It turns out that we do so in the same way as a for loop:

1. We determine the number of steps required to evaluate the leftmost expression in the comprehension. In this case, evaluating `x ** 2` takes 1 step (i.e., is constant time).
2. The collection that acts as the source of the comprehension (in our example, `numbers`), determines how many times the leftmost expression is evaluated.

So let n be the length of the input list `numbers`. The comprehension expression takes n steps (1 step per element of `numbers`). So the running time of `square_all` is n steps, which is $\Theta(n)$. ∎

Importantly, the fact that a comprehension is creating a new collection (in our above example, a list) does *not* count as additional time when analysing the cost of a comprehension. This is true for all three of list, set, and dictionary comprehensions, and so the same analysis would hold in the above function if we had used a set or dictionary comprehension instead.

While loops

Analysing the running time of code involving while loops follows the same principle using for loops: we calculate the sum of the different loop iterations, either using multiplication (when the iteration running time is constant) or a summation (when the iterations have different running times). There is one subtle twist, though: a while loop requires that we write statements to initialize the loop variable(s) before the loop, and update the loop variable(s) inside the loop body. We must be careful to count the cost of these statements as well, just like we did for statements involving loop accumulators in the previous section.

To keep things simple, our first example is a simple rewriting of an earlier example using a while loop instead of a for loop.

Example. Analyse the running time of the following function.

```
def my_sum_v2(numbers: list[int]) -> int:
    """Return the sum of the given numbers."""
    sum_so_far = 0
    i = 0

    while i < len(numbers):
        sum_so_far = sum_so_far + numbers[i]
        i = i + 1

    return sum_so_far
```

Running time analysis. Let n be the length of the input `numbers`.

In this function, we now have both an accumulator and the loop variable to worry about. We can still divide up the function into three parts, and compute the cost of each part separately.

1. The cost of the assignment statements `sum_so_far = 0` and `i = 0` is constant time. We'll count this as a constant-time block of code, which is just *1* step.[1]

2. To analyse the while loop, we need to determine the cost of each iteration and the total number of iterations, just like a for loop.

    - Each iteration is constant time, so we'll count that as one step.
    - There are n iterations, since `i` starts at 0 and increases by 1 until it reaches n. Note that this is less obvious than the for loop version! Here we need to look at three different places in the code: how `i` is initialized, how `i` is updated inside the loop body, and how `i` is used in the loop condition.

3. The return statement again takes constant time, and so counts as 1 step.

[1] This might be a bit surprising, because they are two lines of code and look like two separate "actions". The power of our asymptotic notation is that whether we count this block of code as 1 step or 2, we get the same Theta bound in the end! And so we just go with the simpler one here, but you're welcome to count this as "two steps" in your own analyses if you find that more intuitive.

So the total running time is $1 + n + 1 = n + 2$, which is $\Theta(n)$. ∎

Now, the previous example was a little contrived because we could have implemented the same function more simply using a for loop. Here is another example, which uses a while loop to compute powers of two to act as indexes into a list.

Example. Analyse the running time of the following function.

```
def my_sum_powers_of_two(numbers: list[int]) -> int:
    """Return the sum of the given numbers whose indexes are powers of 2.

    That is, return numbers[1] + numbers[2] + numbers[4] + numbers[8] + ...
    """
    sum_so_far = 0
    i = 1

    while i < len(numbers):
        sum_so_far = sum_so_far + numbers[i]
        i = i * 2

    return sum_so_far
```

Running time analysis. Let n be the length of the input list `numbers`.

This code has much of the same structure as `my_sum_v2`, and we can reuse most of the same analysis here. In particular, we'll still count the initial assignment statements as 1 step, and the return statement as 1 step. To analyse the loop, we still need the number of steps per iteration and the total number of iterations. Each iteration still takes constant time (1 step), same as `my_sum_v2`. It is the number of loop iterations that is most challenging.

To determine the number of loop iterations, we need to take into account the initial value of `i`, how `i` is updated, and how `i` is used in the while loop condition. More formally, we follow these steps:

1. Find a pattern for how `i` changes at each loop iteration, and a general formula formula for i_k, the value of `i` after k iterations. For relatively simple updates, we can find a pattern by writing a small loop tracing table, showing the value of the loop variable at the *end* of the iteration.

Iteration	Value of `i`
0	1
1	2
2	4
3	8
4	16

    So we find that after k iterations, $i_k = 2^k$.[2]

2. We know the while loop continues while `i < len(numbers)`. Another way to phrase this is that the while loop continues *until* `i >= len(numbers)`.

    So to find the number of iterations, we need to find the smallest value of k such that $i_k \geq n$ (making the loop condition False). This is where our formula for i_k comes in:

    $$\begin{aligned} i_k &\geq n \\ 2^k &\geq n \\ k &\geq \log_2 n \end{aligned}$$

    So we need to find the smallest value of k such that $k \geq \log_2 n$. This is exactly the definition of the ceiling function, and so the smallest value of k is $\lceil \log_2 n \rceil$.

[2] Note that we haven't *proved* that this formula is true; a formal proof would require a proof by induction, which you may have already seen in your math classes.

So the while loop iterates $\lceil \log_2 n \rceil$ times, with 1 step per iteration, for a total of $\lceil \log_2 n \rceil$ steps.

Putting it all together, the function `my_sum_powers_of_two` has a running time of $1 + \lceil \log_2 n \rceil + 1 = \lceil \log_2 n \rceil + 2$, which is $\Theta(\log n)$.[3] ∎

[3] Note that our convention is to drop the base of the log when writing a Theta expression, since all bases > 1 are equivalent to each other in Theta bounds.

A trickier example

It turns out that the extreme flexibility of while loops can make analysing their running time much more subtle than it might appear. Our next example considers a standard loop, with a twist in how the loop variable changes at each iteration.

```
def twisty(n: int) -> int:
    """Return the number of iterations it takes for this special loop to stop
    for the given n.
    """
    iterations_so_far = 0
    x = n
    while x > 1:
        if x % 2 == 0:
            x = x / 2
        else:
            x = 2 * x - 2
        iterations_so_far = iterations_so_far + 1

    return iterations_so_far
```

Even though the individual lines of code in this example are simple, they combine to form a pretty complex situation. The challenge with analyzing the runtime of this function is that, unlike previous examples, here the loop variable `x` does not always get closer to the loop stopping condition; sometimes it does (when divided by two), and sometimes it increases!

The key insight into analyzing the runtime of this function is that we don't just need to look at what happens after a single loop iteration, but instead perform a more sophisticated analysis based on *multiple* iterations.[4] More concretely, we'll prove the following claim.

[4] As preparation, try tracing `twisty` on inputs 7, 9, and 11.

Claim. For any integer value of `x` greater than 2, after *two* iterations of the loop in `twisty` the value of `x` decreases by at least one.

Proof. Let x_0 be the value of variable `x` at some iteration of the loop, and assume $x_0 > 2$. Let x_1 be the value of x after one loop iteration, and x_2 the value of x after two loop iterations. We want to prove that $x_2 \leq x_0 - 1$.

We divide up this proof into four cases, based on the remainder of x_0 when dividing by four.[5] We'll only do two cases here to illustrate the main idea, and leave the last two cases as an exercise.

[5] The intuition for these cases is that this determines whether x_0 is even/odd, and whether x_1 is even/odd.

Case 1: Assume $4 \mid x_0$, i.e., $\exists k \in \mathbb{Z},\ x_0 = 4k$.

In this case, x_0 is even, so the `if` branch executes in the first loop iteration, and so $x_1 = \frac{x_0}{2} = 2k$. And so then x_1 is also even, and so the `if` branch executes again: $x_2 = \frac{x_1}{2} = k$.

So then $x_2 = \frac{1}{4}x_0 \leq x_0 - 1$ (since $x_0 \geq 4$), as required.

Case 2: Assume $4 \mid x_0 - 1$, i.e., $\exists k \in \mathbb{Z},\ x_0 = 4k + 1$.

In this case, x_0 is odd, so the `else` branch executes in the first loop iteration, and so $x_1 = 2x_0 - 2 = 8k$. Then x_1 is even, and so $x_2 = \frac{x_1}{2} = 4k$.

So then $x_2 = 4k = x_0 - 1$, as required.

Cases 3 and 4: left as exercises. ∎

Now let's see how take this claim and use it to formally analyse the running time of `twisty`.

Running time analysis. (*Analysis of* `twisty`)

As before, we count the variable initializations before the while loop as 1 step, and the return statement as 1 step.

For the while loop:

- The loop body also takes 1 step, since all of the code consists of operations that do not depend on the size of the input n.

- To count the number of loop iterations, we first observe that x starts at n and the loop terminates when x reaches 1 or less. The *Claim* tells us that after every two iterations, the value of x decreases by at least one.

    So then after 2 iterations, $x \leq n - 1$, after 4 iterations, $x \leq n - 2$, and in general, after $2k$ iterations, $x \leq n - k$. This tells us that after $2(n-1)$ loop iterations, $x \leq n - (n-1) = 1$, and so the loop must stop.

This analysis tells us that the loop iterations *at most* $2(n-1)$ times, and so takes *at most* $2(n-1)$ steps (remember that each iteration takes 1 step).

So the total running time of `twisty` is *at most* $1 + 2(n-1) + 1 = 2n$ steps, which is $\mathcal{O}(n)$. ∎

Something funny happened at the end of the above analysis: we did not actually compute the exact number of steps the function `twisty` takes, only an *upper bound* on the number of steps (signalled by our use of the phrase "at most"). This means that we were only able to conclude a Big-O bound, and not a Theta bound, on the running time of this function: its running time is *at most* $\mathcal{O}(n)$, but we don't know whether this bound is tight.

In fact, it isn't! It is possible to prove something pretty remarkable about what happens to the variable `x` after *three* iterations of the twisty loop.

Claim. (*Improved claim*)

For any integer value of `x` greater than 2, let x_0 be the initial value of `x` and let x_3 be the value of `x` after *three* loop iterations. Then $\frac{1}{8}x_0 \leq x_3 \leq \frac{1}{2}x_0$.

It is a good exercise to prove this claim[6] and then use this claim to conduct a more detailed running time analysis of `twisty`. When you do so, you should be able to show that the running time of `twisty` is both $\mathcal{O}(\log n)$ and $\Omega(\log n)$, and hence conclude that its running time is actually $\Theta(\log n)$, not just $\mathcal{O}(n)$!

[6] Hint: you can use the same approach as the previous claim, but consider remainders when you divide by 8 instead of 4.