Complexity Analysis Part 2[1]

[1] More than just a Theory ...

Robin Dawes

November 14, 2021

Continues our introduction to the use of O-classes to describe the running time of algorithms.

.

For Loops ... Got It. What About While Loops?

So far we have seen how to determine the $O()$ class of an algorithm that contains one or more for loops. We established the technique of assigning the algorithm to the $O()$ class of the highest power of n (the input size) in the time function that we derive by determining how many times each statement is executed. We saw that this process is greatly simplified by identifying the statement or operation that is executed the most often, and then ignoring everything else.

The process can be more difficult with while loops because it may not be immediately obvious how many times the loop will execute.

Sometimes it's easy:

```
read(n)
while (n > 1):
    n = n - 1
    print(n)
```

It's pretty easy to see that this loop executes n-1 times, so the algorithm is $\in O(n)$

But consider this:

```
read(n)
while (n > 1):
    n = n * 0.9
    print(n)
```

When $n = 10$, this loop executes 22 times ... it's not obvious how we could predict that. When $n = 20$, the loop executes 29 times, and

There is a way ... but it involves ... *math!*

when $n = 40$, it executes 36 times. There is a pattern here, and we will come back to it.

And consider this:

```
read(n)
while (n > 1):
    print(n)
```

This either loops 0 times (when the input value is <= 1) or it loops forever! What's the order class for that?

There isn't one!

Let's go back to the second example - the one where n is multiplied by 0.9 on each iteration of the loop. Here's how we can actually figure out the number of times the loop will execute.

I'm going to use n_s to represent the starting value of n After the first iteration, the value of n is $n_s * 0.9$. After the second iteration, n has the value $n_s * 0.9^2$. In general, after the loop iterates i times, n has the value $n_s * 0.9^i$.

The last iteration will be the one that reduces n to <= 1. In other words, the loop will end after the k^{th} iteration when $n_s * 0.9^k \le 1$. Now we just have to figure out k

$$
\begin{aligned}
n_s * 0.9^k &\le 1 \\
n_s &\le \frac{1}{0.9^k} \\
n_s &\le \frac{10}{9}^k \\
\log_{\frac{10}{9}} n_s &\le k
\end{aligned}
$$

In other words, k is the smallest integer $\ge \log_{\frac{10}{9}} n_s$

This may look a bit strange ... we might not be used to seeing a fraction used as the base of the log function, but in fact it is possible use any positive number other than 1 as a logarithmic base. In this example, the necessary base - derived as shown above - is $\frac{10}{9}$

In practical computing problems we almost always work with $\log_2$

$\log_{\frac{10}{9}}(10) = 21.854$ and so the smallest integer ≥ 21.854 is 22 ... which is *exactly* the number of times the loop executes when $n = 10$

Check it out here: `https://www.omnicalculator.com/math/log`

In fact, this while loop always executes $\left\lceil \log_{\frac{10}{9}}(n) \right\rceil$ times, and we can conclude that the algorithm is $\in O(\log_{\frac{10}{9}}(n))$. In fact we can make this simpler. Because of the way logs work, the log of n in any base is just a *constant multiple* of the log of n in any other base. And as we know, O-classification always ignores constant multiples. So when we arrive at a complexity classification like the one we are working with here, we simply ignore the base completely and just write $O(\log n)$.

Here are the general rules for the while loops we have looked at:

If a while loop reduces the value of its test variable by a constant (such as `n = n - 5`) and terminates when the variable drops below a particular value, the loop is in $O(n)$.

If a while loops reduces the value of its test variable by a constant multiplier (such as `n = n / 2`) and terminates when the variable drops below a particular value, the loop is in $O(\log n)$.

$n = n/2$ is the same as $n = n * 0.5$

The bottom line is that while loops are more flexible than for loops and so they can be harder to analyse, but with a bit of effort we can often work it out.

EXERCISE: What is the complexity class of this:

```
read(n)
x = 1
while (x < n):
    x = x*1.3
    print(x)
```

EXERCISE: What is the complexity class of this:

```
read(n)
x = 0
while (x < n):
    x = x + 3
    print(x)
```

Ok, What About Algorithms with Branches?

So far we have seen how to determine the $O()$ class of an algorithm that contains for loops and/or while loops. But most non-trivial algorithms involve some `if` and `else` situations - what about those?

```
read(n)
if n % 3 == 0:
       print(n)
elif n % 3 == 1:
   for i in range(n):
      print(i)
else:
   for i in range(n):
      for j in range(n):
         print(i,j)
```

This algorithm has three possible behaviours - how can we determine its complexity class?

The answer is: O-classification is completely pessimistic - we **always** assume the worst will happen. In terms of classifying the running time of an algorithm, we always assume that the most complex (i.e. highest complexity) branch will be chosen. So the algorithm just above is considered to be $\in O(n^2)$.

This approach[2] makes sense if we are writing software that absolutely has to be fast, all the time. An autonomous car's software module that detects obstacles in the road is useless if it works quickly 75% of the time but is very slow the other 25% of the time. O-classification extends this policy to all algorithms - it puts an upper bound on the worst-case growth rate for the execution time.

[2] which we call WORST-CASE ANALYSIS

In later courses you will study average-case complexity and amortized complexity, which analyze whether or not an algorithm is fast most of the time.

We've talked about the complexity of algorithms containing for loops, while loops and if-else statements. There is one significant type of algorithm we haven't touched yet: recursive algorithms. That's coming up next.