Complexity Analysis Part 3[1]

[1] More than just a Theory ...

Robin Dawes

November 14, 2021

In which we turn our minds to the problem of finding the *O*-classification for recursive algorithms and find the solution to be a method that rewards the ability to recognize patterns.

Recurrence Relations

We have seen how to compute the *O* complexity of programs that involve loops, sequences of operations, and if statements. Now we need to look at recursive functions.

Consider this recursive function:

```
def factorial_rec(n):
    if n == 1:
        return 1
    else:
        return n*factorial_rec(n-1)
```

It clearly makes sense to talk about the number of steps this function executes - there is no randomness or unpredictability involved. Executing a line like

```
x = factorial_rec(3)
```

will always take the same number of steps, and

```
y = factorial_rec(7)
```

will obviously take more steps than

```
x = factorial_rec(6)
```

So there is a relationship between the size of the argument passed to the function and the number of steps that are executed ... but what is the relationship?

We will use $T(n)$ to represent the number of steps factorial_rec(n)

executes. By looking at the definition of factorial_rec(n) we can identify two cases:

1) if $n = 1$, factorial_rec executes a constant number of steps - call it c_1. This lets us state

$$T(1) = c_1$$

2) if $n > 1$, factorial_rec executes a constant number of steps - call it c_2 - followed by a recursive call: factorial_rec(n-1). But this recursive call must take $T(n-1)$ steps (however many that is), so we can state

$$T(n) = c_2 + T(n-1) \qquad \forall\ n > 1$$

Putting these two things together gives us a RECURRENCE RELATION:

$$T(1) = c_1$$

$$T(n) = c_2 + T(n-1) \qquad \forall\ n > 1$$

Note the strong similarity between the form of the recurrence relation, the form of the recursive function, and the form of an inductive proof. They each have a base case and a recursive/inductive part that uses the result for smaller numbers to obtain the result for larger numbers. Induction, recursion and recurrence relations - they all use similar thought patterns - are all part of learning to think like a computer scientist.

That's all well and good, but we need to establish the O complexity of factorial_rec, and that recurrence relation doesn't look anything like the functions we have dealt with before.

We deal with this by transforming the reccurence relation into a closed-form formula : one which does not involve any self-reference. There are many ways to achieve this. We will use one of the most popular, which goes by the name of EXPANSION or SUBSTITUTION. The basic idea is this: we replace the recursive reference to $T(n-1)$ by something of equal value ... but what?

The Substitution Method

Consider rewriting the recursive part of the recurrence relation as $T(x) = c_2 + T(x-1)$. This is clearly still valid - we just changed the n to x. Now let $x = n - 1$, and substitute this into the equation for $T(x)$ - we get

$T(n-1) = c_2 + T(n-1-1)$

i.e.

$T(n-1) = c_2 + T(n-2)$

So we substitute this into $T(n) = c_2 + T(n-1)$ and get

$T(n) = c_2 + c_2 + T(n-2)$

Now we expand $T(n-2)$ to $c_2 + T(n-3)$, and substitute that into the line just above this one, and we get

$T(n) = c_2 + c_2 + c_2 + T(n-3)$

The next expansion gives

$T(n) = c_2 + c_2 + c_2 + c_2 + T(n-4)$

Now we look for a pattern ... and it's pretty easy to spot. When the term inside the $T()$ expression on the right hand side is $n-i$, the number of c_2's is exactly i

So the i^{th} line of the expansion is

$T(n) = c_2 * i + T(n-i)$

Eventually we will get to

$T(n) = c_2 * x + T(1)$ for some x

i.e.

$T(n) = c_2 * x + c_1$

We have successfully eliminated the recursive reference to $T(n-1)$. The question is, what is x?

The final expansion must fit the general pattern, so if the the multiple of c_2 is x, the recursive self-reference must be $T(n-x)$

This gives us $n - x = 1$, which is equivalent to $x = n - 1$

Thus

$T(n) = c_2 * (n-1) + c_1$

But that's something for which we can easily find the Big O classification. Applying what we have already learned, we get

$T(n) \in O(n)$

We will look at several recurrence relations and for each one we will determine its O complexity. You should learn these. Here again is the one we have just seen:

Recurrence Relation	O Classification
$T(1) = c_1$ $T(n) = c_2 + T(n-1)$	$O(n)$

Another Example

Now consider this recursive function (it doesn't do anything except print a lot of numbers, but it is easy to understand)

```
def function_1_rec(n):
    if n == 1:
        return
    else:
        for i in range(n):
            print i
        function_1_rec(n-1)
```

The recurrence relation for function_1_rec looks like this

$T(1) = c_1$

$T(n) = c_2 + c_3 * n + T(n-1)$

Make sure you understand how this recurrence relation is derived from the definition of the function.

We will solve this the same way as we solved the last example, by expanding the recursive reference:

Note that $T(n-1) = c_2 + c_3 * (n-1) + T(n-2)$

$T(n) = c_2 + c_3 * n + c_2 + c_3 * (n-1) + T(n-2)$

$T(n) = c_2 + c_3 * n + c_2 + c_3 * (n-1) + c_2 + c_3 * (n-2) + T(n-3)$

regrouping, we get

$T(n) = c_2 + c_2 + c_2 + c_3 * (n + n - 1 + n - 2) + T(n-3)$

We can identify the pattern now. The i^{th} line of the expansion is

$T(n) = c_2 * i + c_3 * (n + n - 1 + \cdots + n - (i-1)) + T(n-i)$

When we expand this all the way, we get

$T(n) = c_2 + ... + c_2 + c_3 * (n + n - 1 + n - 2 + ...) + T(1)$

As before, we need to work out how many c_2's are in this sum, and now we also have to work out the value of the expression that is multiplied by c_3.

The number of c_2's is easy: as with our previous recurrence relation, the number of c_2's, added to the number in the $T()$ recursive reference, is always n

The value of the expression that is multiplied by c_3 is a little harder to calculate, but we can do it. First we observe that the last element in the sum inside the parentheses is always 1 more than the number in the $T()$ recursive reference. (For example, in the line containing $T(n-3)$, the last element of the sum inside the parentheses is $n-2$.) So when the recursive reference is $T(1)$, the expression inside the parentheses is

$n + n - 1 + n - 2 + ... + 2$

This is a lot like $n + n - 1 + n - 2 + ... + 1$, and we have a formula for that: we know $n + n - 1 + n - 2 + ... + 1 = \frac{(n+1) * n}{2}$

But the left side of this formula is 1 bigger than what we want $(n + n - 1 + n - 2 + ... + 2)$ so we can subtract 1 from each side of the formula to get

$n + n - 1 + n - 2 + ... + 2 = \frac{(n+1) * n}{2} - 1$

Now we can replace all the unknowns in our formula for $T(n)$

$$T(n) = c_2 * (n-1) + c_3 * \left(\frac{(n+1) * n}{2} - 1 \right) + c_1$$

Simplifying this is easy, applying our standard O analysis is even easier, and we end up with

$$T(n) \in O(n^2)$$

Now we have another pattern to add to our collection of recurrence relations:

Recurrence Relation	O Classification
$T(1) = c_1$ $T(n) = c_2 + T(n-1)$	$O(n)$
$T(1) = c_1$ $T(n) = c_2 + c_3 * n + T(n-1)$	$O(n^2)$

Binary Search

Consider the recursive binary search algorithm - it has been posted previously so I won't repeat it here.

The recurrence relation for binary search is

$T(1) = c_1$

$T(n) = c_2 + T(\frac{n}{2})$

Don't just take my word for it! Go through the code and make sure you see why this is true.

Applying our now standard expansion technique, we get

$T(n) = c_2 + c_2 + T(\frac{n}{4})$

$T(n) = c_2 + c_2 + c_2 + T(\frac{n}{8})$

and if we expand this fully we get

$T(n) = c_2 + ... + c_2 + T(1)$

and as always, the question is "how many c_2's are there"? Also, you may be saying that this expansion can't be correct because most of the time n won't divide evenly by 2, 4, 8 etc. Well, you are right, but it turns out not to matter. If you do the recurrence relation with all of the precise details, accounting for odd and even values of n ... you get exactly the same result as we will get by assuming that all the divisions work exactly. So I'm taking the easy route and ignoring those details. Basically, we are assuming that n is a power of 2.

So, back to the question of counting the c_2's. Here we need a bit of clever insight. When the denominator inside the $T()$ recursive reference is 2, there is 1 c_2. When the denominator is 4, there are 2 c_2's. When the denominator is 8, there are 3 c_2's ..... not much of a pattern unless you notice that $2 = 2^1$, $4 = 2^2$, and $8 = 2^3$ .... the number of c_2's is equal to the exponent when the denominator is written as a power of 2.

So when we write

$T(n) = c_2 + ... + c_2 + T(1)$

we need to ask what is the denominator in the $T(1)$ recursive reference.

In other words, what is x, when $\frac{n}{2^x} = 1$?

This is the same as solving for x in $n = 2^x$, and the solution is simply $x = \log_2 n$

As previously explained, we don't need to specify the base of the log when we are working out the O complexity, so we will drop it when we get to the end of this derivation.

Thus the number of c_2's is $\log_2 n$, and we get

$T(n) = c_2 * \log_2 n + c_1$

This doesn't look like the type of function we are used to dealing with in our O classification method but we can handle it exactly the same way. $\log_2 n$ is just a function of n, and it comfortably fills the role of $g(n)$ in the definition of Big-O complexity.

So we conclude that $T(n) \in O(\log n)$ .... and now we have another pattern to add to our table.

Recurrence Relation	O Classification
$T(1) = c_1$ $T(n) = c_2 + T(n-1)$	$O(n)$
$T(1) = c_1$ $T(n) = c_2 + c_3 * n + T(n-1)$	$O(n^2)$
$T(1) = c_1$ $T(n) = c_2 + T(\frac{n}{2})$	$O(\log n)$

Merge Sort

We'll do one more - recursive merge sort. Take time to confirm that the recurrence relation for this recursive function is

$T(1) = c_1$

$T(n) = c_2 + c_3 * n + 2 * T(\frac{n}{2})$

We expand this in the usual way. As with binary search, we make the simplifying assumption that n is a power of 2.

$T(n) = c_2 + c_3 * n + 2 * (c_2 + c_3 * \frac{n}{2} + 2 * T(\frac{n}{4}))$

$T(n) = c_2 * (1 + 2) + c_3 * (n + n) + 4 * T(\frac{n}{4})$

Expand again ...

$T(n) = c_2 * (1 + 2) + c_3 * (n + n) + 4 * (c_2 + c_3 * \frac{n}{4} + 2 * T(\frac{n}{8}))$

$T(n) = c_2 * (1 + 2 + 4) + c_3 * (n + n + n) + 8 * T(\frac{n}{8})$

The final expansion will give us

$T(n) = c_2 * (1 + 2 + 4 + \dots) + c_3 * (n + \dots + n) + x * T(1)$

Now we have to solve $(1 + 2 + 4 + \dots)$

and $(n + \dots + n)$

and x

Inspection shows us that at the i^{th} stage of the expansion, the equation looks like this:

$T(n) = c_2 * (1 + 2 + \dots + 2^{i-1}) + c_3 * i * n + 2^i * T(\frac{n}{2^i})$

As before, we see that when the term inside $T()$ is 1, we must have $n/(2^i) = 1$, which gives us $n = 2^i$, which gives us $i = \log_2 n$

We also see that the number of n's that are multiplied by c_3 is the same i, so in the final expansion the number of n's is $\log_2 n$

And, when $i = \log_2 n$, we get $2^i = n$

This gives

$T(n) = c_2 * (1 + 2 + \dots + 2^{(\log_2 n)-1}) + c_3 * n * \log_2 n + n * T(1)$

That just leaves the $(1 + 2 + \cdots + 2^{(\log_2 n)-1})$ to be resolved. These are powers of 2, and the last term is $2^{(\log_2 n)-1}$

But $2^{(\log_2 n)-1} = \frac{2^{\log_2 n}}{2} = \frac{n}{2}$

So when we get to the end of the expansion, we have $(1 + 2 + 4 + \cdots + \frac{n}{2})$

This is another sum for which we have a simple formula: $1 + 2 + 4 + \cdots + \frac{n}{2} = n - 1$ whenever n is a power of 2.

Putting this in the equation, and replacing $T(1)$ with c_1, we finally get

$$T(n) = c_2 * (n - 1) + c_3 * n * \log_2 n + c_1 * n$$

In this function, the $n * \log_2 n$ term grows the fastest (its growth rate lies between n and n^2), so our standard O-class analysis gives

$$T(n) \in O(n * \log n)$$

And now we can add the final row to our small table of patterns

Recurrence Relation	O Classification
$T(1) = c_1$ $T(n) = c_2 + T(n-1)$	$O(n)$
$T(1) = c_1$ $T(n) = c_2 + c_3 * n + T(n-1)$	$O(n^2)$
$T(1) = c_1$ $T(n) = c_2 + T(\frac{n}{2})$	$O(\log n)$
$T(1) = c_1$ $T(n) = c_2 + c_3 * n + 2 * T(\frac{n}{2})$	$O(n * \log n)$

There are infinitely many possible recurrence relations, but understanding and recognizing these four will get you a long way. You should be able to analyse a recursive function and derive its recurrence relation. If the recurrence relation is one of these four you should be able to state the O classification.

EXERCISE:

Here's another commonly-seen recurrence relation that I could have added to the table. Work out its O classification:

$T(1) = c_1$

$T(n) = c_2 + c_3 * n + T(\frac{n}{2})$