20190912

Determining the Complexity of Recursive Algorithms Using Recurrence Relations

We have seen how to compute the Big-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:

```
A(n):
      if n  >= 1:
            print n
            A(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 = A(3)
```

will always take the same number of steps, and

```
y = A(7)
```

will obviously take more steps than

```
x = A(6)
```

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

Let's define some notation. We will use $T_A(n)$ to represent the number of steps A(n) executes. By looking at the definition of A we can identify two cases:

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

$$T_A(n) = c_1 \text{ for all } n < 1$$

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

$$T_A(n) = c_2 + T_A(n-1) \text{ for all } n \geq 1$$

Putting these two things together gives us something called a recurrence relation:

$$T_A(n) = c_1 \text{ for all } n < 1$$
$$T_A(n) = c_2 + T_A(n-1) \text{ for all } n \geq 1$$

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. Understanding induction, recursion and recurrence relations - they all use similar thought patterns - they are all part of learning to think like a computer scientist.

All well and good, but we need to establish the Big-O complexity of A(n), and that recurrence relation doesn't look anything like the functions we have dealt with before.

We deal with this by transforming the recurrence 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 $T_A(n-1)$ by a different expression that has equal value ... but what?

Consider rewriting the recursive part of the recurrence relation as the more generic equation $T_A(x) = c_2 + T_A(x-1)$. This is clearly still valid - we just changed the n to x. I'm calling it more generic because it is stepping away from the particular value of n that we started with, and replacing it with a generic place-holder x. Now let x = n-1, and substitute this into the equation for $T_A(x)$. We get

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

ie

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

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

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

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

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

The next expansion gives

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

Eventually we will get to

$$T_A(n) = c_2 + \cdots + c_2 + T_A(0)$$

ie

$$T_A(n) = c_2 + \cdots + c_2 + c_1$$

We have successfully eliminated the recursive reference to $T_A()$. The question is, how many c_2's are there in this expression?

It's not hard to see that at each stage of the expansion, the number of c_2's plus the number inside the $T_A()$ recursive reference always equals n. (For example, the first line has $T_A(n) = 1 * c_2 + T_A(n-1)$. The next line has $T_A(n) = 2 * c_2 + T_A(n-2)$. Then we get $T_A(n) = 3 * c_2 + T_A(n-3)$ etc.) So in the line $T_A(n) = c_2 + \cdots + c_2 + T_A(0)$ there must be n c_2's, so we can write

$$T_A(n) = c_2 * n + c_1$$

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

$T_A(n)$ is in O(n)

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

$$T_A(n) = c_1 \text{ for all } n < 1$$
$$T_A(n) = c_2 + T_A(n-1) \text{ for all } n \geq 1$$

Recurrence Relation	Big O Complexity
$T_A(0) = c_1$ $T_A(n) = c_2 + T_A(n-1)$	O(n)

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

```
B(n):
    if n >= 1:
        for i in range(n):
            print i
        B(n-1)
```

The recurrence relation for the time function of B(n) looks like this

$T_B(n) = c_1$ for n < 1

$T_B(n) = c_2 + c_3 * n + T_B(n-1)$ for $n \geq 1$

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

We solve this the same way as before. Note that

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

so

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

and then

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

regrouping, we get

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

Note that the last term in the expression that is multiplied by c_3 is always 1 greater than the value in the recursive reference to T_B (for example, when the multiple of c_3 is n + n-1, the value in the recursive reference is n-2)

Thus when we expand this all the way, we get

$$T_B(n) = c_2 + \cdots + c_2 + c_3 * (n + n - 1 + n - 2 + \cdots + 1) + T_B(0)$$

$$T_B(n) = c_2 + \cdots + c_2 + c_3 * (n + n - 1 + n - 2 + \cdots + 1) + c_1$$

Once again 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 $T_B()$ 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. The sum n + n-1 + n-2 + ... + 1

works out to $\frac{(n+1) * n}{2}$

(If this is not already familiar, it is not hard to prove in a variety of ways – induction is particularly easy in this case.)

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

$$T_B(n) = c_2 * n + c_3 * \frac{(n+1) * n}{2} + c_1$$

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

$T_B(n)$ is in $O(n^2)$

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

Recurrence Relation	Big O Complexity
$T(0) = c_1$ $T(n) = c_2 + T(n-1)$	O(n)
$T(0) = c_1$ $T(n) = c_2 + c_3 * n + T(n-1)$	O(n^2)

Now consider the recursive binary search algorithm - it is well known so I won't repeat it here.

The recurrence relation for binary search is

$T_{BS}(1) = c_1$
$T_{BS}(n) = c_2 + T_{BS}\left(\frac{n}{2}\right)$ for n > 1

Note that the base case here is for n = 1 instead of 0 as in the previous problems. It turns out this makes no difference at all. The base case can actually use any specific value without changing the final complexity analysis.

DON'T JUST TAKE MY WORD FOR IT. MAKE SURE YOU SEE WHY THIS IS TRUE.

Applying our now standard expansion technique, we get

$$T_{BS}(n) = c_2 + c_2 + T_{BS}\left(\frac{n}{4}\right)$$

$$T_{BS}(n) = c_2 + c_2 + c_2 + T_{BS}\left(\frac{n}{8}\right)$$

and if we expand this fully we get

$$T_{BS}(n) = c_2 + \cdots + c_2 + T_{BS}(1)$$

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 this doesn't 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.

So, back to the question of counting the c_2's. Here we need a bit of clever insight. When the denominator inside the $T_{BS}()$ 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

$$T_{BS}(n) = c_2 + \cdots + c_2 + T_{BS}(1)$$

we need to ask what is the denominator in the $T_{BS}()$ 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 $x = \log n$ (the log is to the base 2, but we don't have to worry about that detail either)

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

$$T_{BS}(n) = c_2 * \log n + c_1$$

This doesn't look like the type of function we are used to dealing with in our Big-O complexity analysis, but we can handle it exactly the same way. Log 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) is in O(log n) .... and now we have another pattern to add to our table.

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

One more recursive function pattern. This one corresponds to many algorithms, including mergesort which we will look at next day. For now, we will just look at the recurrence relation.

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

We expand this in the usual way

$$T_{MS}(n) = c_2 + c_3 * n + 2 * \left(c_2 + c_3 * \frac{n}{2} + 2 * T_{MS}\left(\frac{n}{4}\right)\right)$$

$$T_{MS}(n) = c_2 * (1 + 2) + c_3 * (n + n) + 4 * T_{MS}\left(\frac{n}{4}\right)$$

Expand again ...

$$T_{MS}(n) = c_2 * (1 + 2) + c_3 * (n + n) + 4 * \left(c_2 + c_3 * \frac{n}{4} + 2 * T_{MS}\left(\frac{n}{8}\right)\right)$$

$$T_{MS}(n) = c_2 * (1 + 2 + 4) + c_3 * (n + n + n) + 8 * T_{MS}\left(\frac{n}{8}\right)$$

… and again until the final expansion will give us

$$T_{MS}(n) = c_2 * (1 + 2 + 4 + ...) + c_3 * (n + ... + n) + x * T_{MS}(1)$$

Now we have to solve

(1+2+4+...)

and (n+....+n)

and x

Inspection shows us that at every stage of the expansion, the term inside the $T_{MS}()$ recursive reference is of the form $\frac{n}{2^i}$, and the coefficient of $T_{MS}()$ is also 2^i. As before, we see that

when the term inside $T_{MS}()$ is 1, we must have $\frac{n}{2^i} = 1$, which gives us $n = 2^i$, which gives us $i = \log n$

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 $\log n$

This gives

$$T_{MS}(n) = c_2 * (1 + 2 + 4 + ...) + c3 * n * \log n + n * T_{MS}(1)$$

That just leaves the (1+2+4+...). These are powers of 2, and the last one is always 1/2 the coefficient of the $T_{MS}()$ recursive reference. So when we get to the end of the expansion, we have
(1 + 2 + 4 + ... + $\frac{n}{2}$)

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

Putting this in the equation, and replacing $T_{MS}(1)$ with c_1, we finally get

$$T_{MS}(n) = c_2 * (n - 1) + c_3 * n * \log n + c_1 * n$$

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

$T_{MS}(n)$ is in O($n * \log n$)

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

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

There are infinitely many possible recurrence relations, but these four will cover the vast majority of recursive functions that you will encounter in the real world. 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 Big O complexity.