Big-Oh for Recursive Algorithms

Often when analyzing recursive algorithms, the running time of the algorithm will be defined in terms of itself. A recursive algorithm always has at least one recursive case, where the algorithm calls itself with a smaller-sized input, and a base case, where the algorithm does not call itself recursively. When we analyze this situation to determine a running time function $T(n)$, the definition of $T(n)$ will also have (at least two) components, a recursive one where $T(n)$ is defined in terms of itself, and a base case, where $T(n)$ is not defined in terms of itself.

For instance, the binary search algorithm has a recursive case and a base case. The base case occurs when the input size is one (an array with one element), and the algorithm does some fixed amount of work (to determine if the single element in the array is equal to the item we are searching for). So we can define $T(1) = 1$ here. For the recursive case, binary search does some fixed amount of work (to find the middle element of the array and examine its value) and then calls itself recursively on (roughly) half of the array. So we can define $T(n) = T(\frac{n}{2}) + 1$, for $n \geq 1$.

Summarizing, for binary search,

$$T(n) = \begin{cases} T(\frac{n}{2}) + 1 & \text{if } n \geq 1 \\ 1 & \text{if } n = 1. \end{cases}$$

Once you have turned your algorithm's code into a recursive formula for $T(n)$, the next step is to solve the recurrence (turn the recursive formula into a non-recursive one). We will do that in the series of steps below.

Step 1: Find a pattern in recursive calls to $T(n)$

We do this via the *telescoping method.* Basically, this method asks us to find a pattern in the recursion to rewrite $T(n)$ not in terms of a smaller version of itself.

First, write the formula for $T(n)$ a few times, substituting in the smaller input size each time:

$$T(n) = T(\tfrac{n}{2}) + 1 \tag{1}$$

Now substitute $n/2$ back into the formula on the left for the original n:

$$T(\tfrac{n}{2}) = T(\tfrac{n}{4}) + 1 \tag{2}$$

And continue to do this:

$$T(\tfrac{n}{4}) = T(\tfrac{n}{8}) + 1 \tag{3}$$
$$T(\tfrac{n}{8}) = T(\tfrac{n}{16}) + 1 \tag{4}$$
$$T(\tfrac{n}{16}) = T(\tfrac{n}{32}) + 1 \tag{5}$$

Once you are comfortable with this, and see the general pattern, we return to the original recursive formula and rewrite only the right side of that formula over and over, substituting in equations as we go:

$$T(n) = T(\tfrac{n}{2}) + 1$$

Now substitute the entire $T(\frac{n}{2})$ formula on the right side above with its definition that you wrote down earlier (from Equation 2):

$$T(n) = T(\tfrac{n}{2}) + 1 = \big(T(\tfrac{n}{4}) + 1) + 1 = T(\tfrac{n}{4}) + 2$$

Now substitute in Equation 3, and continue for subsequent equations:

$$\begin{aligned} T(n) &= T(\tfrac{n}{4}) + 2 = \big(T(\tfrac{n}{8}) + 1\big) + 2 = T(\tfrac{n}{8}) + 3 \\ T(n) &= T(\tfrac{n}{8}) + 3 = \big(T(\tfrac{n}{16}) + 1\big) + 3 = T(\tfrac{n}{16}) + 4 \\ T(n) &= T(\tfrac{n}{16}) + 4 = \big(T(\tfrac{n}{32}) + 1\big) + 4 = T(\tfrac{n}{32}) + 5 \end{aligned}$$

The goal of this process is to find a pattern that relates to the number of times we have performed this substitution. Here, notice how every time we substitute, the denominator of the fraction inside T doubles (from 2 to 4 to 8, etc), and the constant at the end of the formula increases by one (from 1 to 2 to 3, etc).

We want to answer the question, "After k steps of substitution, what does the formula look like?" To answer this question, consider it separately for each part of the formula that is changing. For the denominator of the fraction, the pattern is *exponential*: after k steps, the denominator is 2^k. For the constant at the end, the pattern is *linear*: after k steps, the constant is k.

So the general formula is

$$T(n) = T(\tfrac{n}{2^k}) + k \tag{6}$$

In particular, notice how if we substitute $k = 1, 2, 3, \ldots$ in the equation above, we obtain the exact equations we derived earlier.

Step 1 is finished when we have a formula for $T(n)$ in terms of not only n, the input size, but also k, the number of steps of substitution we have done.

Step 2: Figure out when the recursion stops

Once we have a general formula for $T(n)$ in terms of the number of steps k, we need to figure out when the recursion stops. In other words, how many steps k will the substitution process above last before we hit the base case?

This is straightforward to figure out. Because Equation 6 tells us $T(n) = T(\frac{n}{2^k}) + k$ and the base case is reached when we get to $T(1)$, let's just set $\frac{n}{2^k} = 1$. We can manipulate this equation as follows:

$$\frac{n}{2^k} = 1 \qquad \Leftrightarrow \qquad n = 2^k \qquad \Leftrightarrow \qquad \log_2 n = k$$

This implies that the recursion/substitution lasts $k = \log n$ steps. (In computer science, if the base of a logarithm is unspecified, base-2 is often assumed since it appears more often than base-10 or base-e (the natural logarithm)).

Step 3: Substitute for k back into the $T(n)$ formula

Our goal now is to return to our recursive formula for $T(n)$ from the end of step 1, and eliminate all references to k (by equivalent references to n).

At the end of step 1, we learned $T(n) = T(\frac{n}{2^k}) + k$ (Equation 6). However, we now also know $\frac{n}{2^k} = 1$, $n = 2^k$, and $\log_2 n = k$ (all of these are of course equivalent). Use any or all of these to get rid of any references to k in the $T(n)$ equation:

$$\begin{aligned}T(n) &= T(\tfrac{n}{2^k}) + k\\ T(n) &= T(1) + \log_2 n\end{aligned}$$

and now substitute $T(1) = 1$:

$$T(n) = 1 + \log_2 n.$$

Step 4: Determine big-oh from the non-recursive $T(n)$ formula

The last step is easy. Once we have a non-recursive formula for $T(n)$, use all the regular big-oh techniques to determine the big-oh complexity.

If $T(n) = \log_2 n + 1$, then $T(n) = O(\log_2 n)$. This illustrates how binary search is a logarithmic-time algorithm.

Another example

Let's repeat this process with another example, the standard recursive algorithm for computing the Fibonacci sequence. The Fibonacci sequence is defined as $F(n) = F(n-1)+F(n-2)$ for $n \geq 2$, with $F(0) = F(1) = 1$. This leads to a (naive) recursive formulation in pseudocode:

```
function fib(int n)
  if n >= 1:
    return 1
  else:
    return fib(n-1) + fib(n-2)
```

In the recursive case, we are recursing *twice*, plus doing a constant amount of work, so we have $T(n) = T(n-1) + T(n-2) + 1$. In the base case, we just do a constant amount of work, so $T(0) = T(1) = 1$.

Unfortunately, this is a rather complicated recurrence to solve because of the two different base cases, so we will simplify a little bit. Let's pretend that Fibonacci recurses twice on

$n-1$ (rather than once on $n-1$ and once on $n-2$) — this will make our lives much easier. Therefore, we can simplify the recursive case to $T(n-1)+T(n-1)+1 = 2T(n-1)+1$. So our recursive formulation becomes:

$$T(n) = \begin{cases} 2T(n-1)+1 & \text{if } n \geq 1 \\ 1 & \text{if } n = 1. \end{cases}$$

Let's walk through the same steps as before.
Step 1: Find a recursive pattern in $T(n)$:

$$\begin{aligned} T(n) &= 2T(n-1)+1 \\ T(n-1) &= 2T(n-2)+1 \\ T(n-2) &= 2T(n-3)+1 \\ T(n-3) &= 2T(n-4)+1 \\ &\text{(pattern is clear)} \end{aligned}$$

Step 1.5: Substitute each equation back into $T(n)$:

$$\begin{aligned} T(n) &= 2T(n-1)+1 && = 2T(n-1)+1 \\ &= 2\left[2T(n-2)+1\right]+1 \\ &= 4T(n-2)+2+1 && = 4T(n-2)+3 \\ &= 4\left[2T(n-3)+1\right]+2+1 \\ &= 8T(n-3)+4+2+1 && = 8T(n-3)+7 \\ &= 8\left[2T(n-4)+1\right]+4+2+1 \\ &= 16T(n-4)+8+4+2+1 && = 16T(n-4)+15 \end{aligned}$$

Now we try to find the pattern in each part of the formula that changes from step to step. There are three parts that are changing: the coefficient in front of the T is following an exponential sequence $2, 4, 8, 16, \ldots$, whereas the part inside the parentheses is following a linear sequence $n-1$, $n-2$, $n-3$, etc. But the part after the plus sign is harder to recognize (the pattern involving 1, 2+1, 4+2+1, etc).

There are two ways to simplify this, both mathematically equivalent. The first is to realize that the sequence 1, 2+1, 4+2+1, etc, is a geometric series:

$$\sum_{i=0}^{p} 2^i = 2^{p+1} - 1.$$

One can also notice that the number after plus sign is just the coefficient before the T with one subtracted from it.

At this point, we can introduce the number of steps k to obtain:

$$T(n) = 2^k T(n-k) + (2^k - 1)$$

Step 2: Find where the recursion stops:

In our original $T(n)$ definition, we know the base case is $T(1) = 1$. So with our equation $T(n) = 2^k T(n-k) + (2^k - 1)$, this will stop recursing when $n - k = 1$.

$$n - k = 1 \qquad \Leftrightarrow \qquad n = k + 1 \qquad \Leftrightarrow \qquad k = n - 1$$

Step 3: Substitute back in:

$$T(n) = 2^k T(n-k) + (2^k - 1)$$
$$T(n) = 2^{n-1} T(1) + (2^{n-1} - 1)$$

and now substitute $T(1) = 1$:

$$T(n) = 2^{n-1} + (2^{n-1} - 1) = 2^n - 1$$

Step 4: Determine big-oh

$T(n) = 2^n - 1 = O(2^n)$. So the naive Fibonacci algorithm takes exponential time. Note that there are other ways to calculate the Fibonacci sequence that are much faster — for instance, in $O(n)$ (linear) time.

Exercises

Calculate a non-recursive $T(n)$ and a big-oh complexity for the following:

1. $T(n) = T(n/2) + n; T(1) = 1$
2. $T(n) = 2T(n/2) + 1; T(1) = 1$
3. $T(n) = 2T(n/2) + n; T(1) = 1$
4. $T(n) = T(n-1) + 1; T(1) = 1$
5. $T(n) = T(n-1) + n; T(1) = 1$
6. $T(n) = 2T(n-1) + n; T(1) = 1$