Recursive algorithms and their complexities

Carson Mulvey

When it comes to recursive algorithms, it can be difficult to figure out time complexity. Consider the following basic example:

```
def procedure1(n):
    if (n == 1):
        return 1
    else:
        return procedure1(n-1) + 1
```

It helps to first understand what the procedure actually does. See that each output of `procedure1` is 1 plus the previous output, `procedure1(1)` returning 1. That is, `procedure1(2)` returns 2, `procedure1(2)` returns 3, et cetera. Therefore, we know that `procedure1(n)` returns n.

Now let's find the time complexity. See how the function first tests if $n = 1$, which is constant time, or $O(1)$. When $n \neq 1$, we call `procedure1` with the previous n, and add 1 to that result. Thus, ignoring the recursive call, our procedure is $O(1)$.

Now lets see how many times the procedure is called. We see that `procedure1(n)` calls `procedure1(n-1)`, which then calls `procedure1(n-2)`, and so on, until we arrive at `procedure1(1)`. Because of this, we end up with n procedure calls, each of which are $O(1)$. Therefore, our time complexity is $O(n \cdot 1) = O(n)$.

Let's take another example:

```
def procedure2(n):
    if (n == 0):
        return 1
    elif (n == 1):
        return 2
    else:
        return procedure2(n-2) * 4
```

Note how this algorithm uses $n - 2$ instead of $n - 1$ in the recursive definition. In the algorithm, we multiply the second-previous term by 4, which overall leads to `procedure2(n)` outputting 2^n (try out numbers to see this yourself!).

Now for time complexity. Like before, everything besides the recursive call is $O(1)$. Then since `procedure2(n)` calls `procedure2(n-2)`, which then calls `procedure2(n-4)`, and so on, until we arrive at `procedure1(1)` or `procedure1(0)`. Since we decrease by 2 each time, this will be about $n/2$ calls. Thus, the complexity is $O(n/2 \cdot 1) = O(n)$.

See how the complexity is $O(n)$ for both `procedure1` and `procedure2`! This applies to any simple[1] recursion where n calls $n - c$ for some positive integer c and calls nothing else.

Here's another example:

```
def procedure3(n):
    if (n == 0):
        return 0
    else:
        return procedure3(n//2) + 1
```

First note that ignoring recursion, a call of the procedure is $O(1)$. Each procedure recursively calls itself with half the input (with floor division). For example, `procedure3(20)` calls `procedure3(10)`, which calls `procedure3(5)`, then `procedure3(2)`, then `procedure3(1)`, and finally `procedure3(0)`. Noting that the 'amount of times n can be divided by 2 before reaching 0' is about $\log_2 n$, we conclude that the procedure is $O(\log n)$. This applies to any simple recursion where n calls n/c for some positive integer c.

[1]By simple, I mean that the procedure recursively calls itself once, with the rest of the function running in constant time

Onto the next example:

```
def procedure4(n):
    sum = 0
    for i in range(n):
        sum += 1
    if (n == 0):
        return 0
    else:
        return procedure4(n-1) + sum
```

Analyzing this algorithm, we see that there is a for loop with $O(n)$ time complexity. Thus, ignoring the recursion, `procedure4` has linear time complexity. Because the procedure calls the previous term, there will be a total of n calls. Thus, our time complexity is $O(n \cdot n) = O(n^2)$.

In general, if a recursive function has a $O(f(n))$ complexity *besides* the recursive call, then $f(n)$ is multiplied to the amount of calls that occur.

Note: The rest of this document is beyond course content

In our discussion, we covered Problem 5.4.29, which had the following solution:

```
def a(n):
    if (n == 0):
        return 1
    elif (n == 1):
        return 2
    else:
        return a(n-1) * a(n-2)
```

I stated in class that at first glance, the time complexity looked like $O(n^2)$, but this was *wrong*!

A key difference between `a` and the previous algorithms is that *multiple* recursive calls are being made. To figure out how many calls are made, we'll let calls(n) be the number of total function calls for `a(n)`. Then since `a(0)` and `a(1)` are base cases, calls(0) = calls(1) = 1. Also, because `a(n)` calls `a(n-1)` and `a(n-2)`, we have calls(n) = calls($n-1$) + calls($n-2$) (looks familiar?).

Because of this, we actually get $\text{calls}(n) = \text{fib}_{n+1}$, and because the rest of the algorithm is constant, that gives a time complexity of $O(\text{fib}_n)$. The explicit formula for the fibonacci numbers is not required knowledge for this class, but using it, we can also write the complexity as $O(\varphi^n)$, where $\varphi \approx 1.61803$ is the golden ratio. Pretty neat!