15-150 Lecture 5: Asymptotic Analysis

Lecture by Dan Licata

January 31, 2012

In this lecture, we introduce the idea of *asymptotic analysis*. This includes the following concepts:

- recurrence relations
- closed forms
- big-O notation
- time analysis of functions

1 Reverse

Let's say I have a list of things, like this deck of science fiction author playing cards. I want to reverse the list, so that instead of having my favorite authors on top, they're on the bottom (save the best for last). How do I do that?

Let's start with the template, assuming we'll do structural recursion:

```
(* Purpose: reverse [x1,x2,...,xn] == [xn,xn-1,...,x2,x1] *)
fun reverse (l : int list) : int list =
    case l of
        [] => ...
      | x :: xs => ... (reverse xs) ...
val [3,2,1] = reverse [1,2,3]
```

At this point, if you're stuck a good idea is to do some examples: what should the reverse of `[]` be? According to the spec, it needs to be all the elements of `[]` in the opposite order, which is `[]`. So we can fill in the base case:

```
fun reverse (l : int list) : int list =
    case l of
        [] => []
      | x :: xs => ... (reverse xs) ...
```

Now if you're stuck on the cons case, a good technique is to *do an example where you assume the recursive call works:*

```
reverse [1,2,3,4] should evaluate to [4,3,2,1] by the spec
reverse [2,3,4] should evaluate to [4,3,2] if the recursive call works
```

So the question is how to get from `[4,3,2]` to `[4,3,2,1]`. And the answer is: put `1` at the back! We can't do this using `::`, which adds something to the front of a list. But we can using a new operation `@` (append), which combines two lists in order; e.g. `[1,2,3]@[4,5,6] == [1,2,3,4,5,6]`. Thus, the final code is:

```
fun reverse (l : int list) : int list =
    case l of
        [] => []
      | x :: xs => (reverse xs) @ [x]
```

How do you define append? It's built-in, but it would be defined like this:

```
fun (l1 : int list) @ (l2 : int list) : int list =
    case l1 of
        [] => l2
      | x::xs => x :: (xs @ l2)
```

Append recurs down `l1` and creates a new list where the `[]` from `l1` is replaced with `l2`.

2 Time Complexity

One way to choose between different algorithms for the same problem is to analyze their *time complexity*. What this means is that you look at the code and extract a prediction about how long it will take to run, as a function of the size of the input. The whole point is to predict how long a function will take on big inputs, before you actually run it—so you know whether it will be fast enough or not.

We break time complexity analysis down into three steps:

- Go from the code to a *recurrence relation*
- Go from the recurrence relation to a *closed form*
- Go from the closed form to a *big-O bound*

2.1 Recurrence Relations

Let's use append as a first example. How long does it take to run `l1 @ l2`?

- The zero case takes 2 steps:

```
    [] @ l2
|-> case [] of [] => l2 | ...
|-> l2
```

- The `x::xs` case takes 2 steps plus the time for the recursive call:

```
    (x::xs)@l2
|-> case x::xs of ... | x::xs => x::(xs@l2)
|-> x::(xs@l2)
|-> ... |-> ... x::v (recursive call computes a value v)
```

This analysis exploits the fact that there is a well-defined, mathematical, notion of *calculation step* in functional programming. For this reason, computing by calculation is good both for verification (as we saw when we used it in proofs last time) and for analysis of algorithms.

We can summarize this by defining a *recurrence relation* $W_{@}(n)$ representing the number of steps it takes to run `l1 @ l2` where `l1` has length n. Note that the length of `l2` is irrelevant, because the number of steps that we counted above is independent of the length of `l2`. Here's the recurrence:

$$W_{@}(0) = 2$$
$$W_{@}(n) = 2 + W_{@}(n-1) \text{for non-zero } n$$

"Recurrence relation" is just a fancy name for a recursive definition of a function in math (not in SML). It's easy to extract a recurrence relation from a functional programming: you count the steps it takes to get to the recursive call in each case. However, a recurrence relation is not a very useful tool for comparing different algorithms, because the recurrence relation has the same recursion structure as the code itself, but different algorithms might have different recursion structure.

2.2 Closed Forms

A more useful thing to consider is a *closed form* for the recurrence relation: a non-recursive specification of the same function. For example:

Theorem 1. *For all n, $W_{@}(n) = 2n + 2$.*

In this case, the *closed form* of $W_{@}$ is $2n + 2$. Why? Well, you can see that $W_{@}$ adds 2 for every 1 that it peels off, with a base case of 2. This is just like the correctness of `double` that you're doing in the homework. Formally, we can prove this using induction:

Proof. **Case for 0:** To show: $W_{@}(0) = 2 * 0 + 2$. True by calculation.

Case for $1 + k$: Inductive hypothesis: $W_{@}(k) = 2 * k + 2$.
To show: $W_{@}(1 + k) = 2 * (1 + k) + 2$. Proof: By definition, $W_{@}(1 + k) = 2 + W_{\texttt{exp}}(k)$. By the IH, this equals $2 + 2 * k + 2$ which equals $2 * (1 + k) + 2$. □

The closed form is a better representation for comparison, because it abstracts away from the particular recursion pattern used to define the function.

2.3 Big-O Notation

Big-O notation is a way to abstract away a little more. The idea with big-O notation is to give a "rough" upper bound on the value of a function.

To a first approximation, $fisO(g)$ means that f is bounded from above by g. So, to show that $fisO(g)$, it suffices to show that f is always less than g—i.e. for all x, $f(x) \leq g(x)$. For example $xisO(x^2)$.

However, for comparing the running times of algorithms, it is often useful to ignore a couple of things. First of all, we will *ignore constant factors*. For example, $6xisO(2x)$, even though $6x$ is never less than or equal to $2x$. To show that f is $O(g)$, it suffices to show that there exists a k such that for all x, $f(x) \leq k * g(x)$. That is, f must be less than some constant multiple of g.

Second, we will ignore what happens on small inputs. For example, $x + 6isO(x^2)$, even though $x + 6$ is bigger up to a point. Thus, f is $O(g)$ if there is some point after which f is always less than g. Putting these two together, we get the following definition:

> f is $O(g)$ iff there exist k, x_0 such that for all $x \geq x_0$, $f(x) \leq k * g(x)$.

In the example, $W_{\texttt{@}}(n)$, which equals $2n + 2$, is $O(n)$. We say that `@` runs in *linear time*.

O-notation is useful for analyzing the time complexity of algorithms because it gives you a rough picture of how the running time depends on the input size, on large inputs—i.e. it gives you a rough picture of the *asymptotic analysis* as the input size approaches infinity. For example, an $O(n)$ algorithm for a problem will usually be faster on large inputs than an $O(n^2)$ algorithm. However, you can hide a lot in k and x_0: sometimes an algorithm with better big-O has such large constants that it is worse in practice. But in many case asymptotic analysis gives you good predictions about which algorithm would run better on big inputs.

In this course, we will ask you to extract recurrence relations from your code, compute closed forms, and use them to determine the asymptotic big-O time complexity of a function. Here wre some big-O's we'll run into:

$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential

and so on.

Each of these functions in the above list is O(the ones below it). E.g. n is $O(n^2)$, and $O(2^n)$. Thus, it's important to keep in mind that saying "f is $O(g)$" does not mean that g is the best upper bound for f. None of the functions are O(the ones above them). E.g. there is no constant multiple of $\log n$ that is (eventually) always greater than n.

When we ask you for the O of a function or recurrence relation, we expect you to give a *tight bound*: i.e. one where f is bounded both above and below by g. (Otherwise you could always say $O(2^n)$ and almost always be right.) Technically, the notation for this is "f is $\Theta(g)$", but we will be sloppy and use O.

2.4 Reverse

Let's do the same thing for reverse:

Recurrence Here's a recurrence for the work of reverse:

$$W_{\texttt{reverse}}(0) = k_0$$
$$W_{\texttt{reverse}}(n) = k_1 + W_{\texttt{@}}(n-1) + W_{\texttt{reverse}}(n-1)$$

There are a few things to note: First, if you're doing big-O analysis at the end, the constants don't matter. Thus, you can make your life easier when writing recurrences by not calculating out exactly how many steps something takes, but instead writing $k_0, k_1, \ldots$ for some fixed constant number of steps. Second, the recurrence for `reverse` refers to the recurrence for `@`, just like the code does. Third, it's actually a little subtle to figure out what the argument to $W_{@}$ should be: it's the length of the first argument to `@`, which is `reverse xs`. `xs` has length $n-1$, and it's a simple theorem that the length of `reverse xs` is the same as the length of `xs`. Formally, we should prove this, but when doing analysis we won't always prove these relationships explicitly.

Closed Form Next, we want to compute a closed form for the recurrence. The first step is to plug in the closed form for $W_{@}$:

$$\begin{aligned} W_{\texttt{reverse}}(n) &= k_1 + 2 * (n-1) + 2 + W_{\texttt{reverse}}(n-1) \\ &= k_1 + 2n + W_{\texttt{reverse}}(n-1) \\ &= k_1 + k_2 n + W_{\texttt{reverse}}(n-1) \end{aligned}$$

and replace the 2 by an abstract constant.

What is the closed form of a recurrence of this form? To see this, it is helpful to *expand* the recurrence and rewrite it as a sum:

$$\begin{aligned} W_{\texttt{reverse}}(n) &= k_1 + k_2(n-1) \\ &\quad +k_1 + k_2(n-2) \\ &\quad +k_1 + k_2(n-3) \\ &\quad \ldots \\ &\quad +k_0 \end{aligned}$$

Thus what we have is

$$W_{\texttt{reverse}}(n) = \textstyle\sum_{i=0}^{n-1}(k_1 + k_2 i)$$

Then we can use properties of arithmetic and sums to get a closed form:

$$\begin{aligned} \textstyle\sum_{i=0}^{n-1}(k_1 + k_2 i) &= k_1 * (\textstyle\sum_{i=0}^{n-1} 1) + k_2 * (\textstyle\sum_{i=0}^{n-1} i) + k_0 \\ &= k_1 * (n-1) + k_2 * (\tfrac{(n-1)n}{2}) + k_0 \\ &= k_3 * n + k_4 n^2 + k_5 \end{aligned}$$

In the second step, we used the equations

$$\begin{aligned} (\textstyle\sum_{i=0}^{n} 1) &= n \\ (\textstyle\sum_{i=0}^{n} i) &= \tfrac{n(n+1)}{2} \end{aligned}$$

In the third, we did some arithmetic and coalesced constants into some new constants k_3 and k_4 (it's not hard to work out what k_3 and k_4 and k_5 are in terms of k_1 and k_2 and k_0).

Thus, we arrive that the closed form

$$W_{\texttt{reverse}}(n) = k_5 + k_3 * n + k_4 n^2$$

big-O This closed form is $O(n^2)$. In general, when you have a sum of terms, the big-O is the biggest summand.

Thus, `reverse` takes quadratic time. The reason is that `@` takes linear time, and we do one append for each element of the list. It's not quite as simple as this explanation makes it seem, because the list we are appending onto is smaller each time. But the formula for the closed form of n summorial (cf. your HW 2 problem) tells us that this is still quadratic.

3 Fast Reverse

However, there's a faster way to reverse a list: just place the elements into a separate pile, one by one (e.g. watch me reverse this deck of science fiction author playing cards). The second pile will then be the first pile in reverse order.

To implement this, we generalize the function so that it takes a second argument, representing the second pile:

```
(* Purpose: reverse l in linear time *)
fun revTwoPiles (l : int list, r : int list) : int list =
    case l of
        [] => r
      | (x :: xs) => revTwoPiles(xs , x :: r)

fun fastReverse (l : int list) : int list = revTwoPiles(l , [])
```

In the `x::xs` case of `revTwoPiles`, we pick up one card from the left pile, put it on the right, and then continue the process by recurring. `fastReverse` calls `revTwoPiles` with the initial right pile empty.

Analysis: The recurrence is

$$W_{\texttt{revTwoPiles}}(0) = k_0$$
$$W_{\texttt{revTwoPiles}}(n) = k_1 + W_{\texttt{revTwoPiles}}(n-1)$$

$$W_{\texttt{fastReverse}}(n) = k_2 + W_{\texttt{revTwoPiles}}(n)$$

In the nil case, the function does a constant number of steps. In the cons case, the function does a constant number of steps plus the recursive call. Analogously to append, the closed form is $k_0 + k_1 * n$, so the work is $O(n)$. Thus, this version takes linear time.

This is our first example illustrating the idea that *harder problems can be easier*: by generalizing the problem statement, we were able to get a faster algorithm.

4 Additional Example: Fast Exponentiation

Here is another example of analysis (this wasn't covered in lecture, but might be helpful).

4.1 Analysis of `exp`

Recall `exp` from last week:

```
(* Purpose: compute 2^n
   Examples: exp 0 ==> 1
             exp 4 ==> 16
*)
fun exp (n : int) : int =
    case n of
        0 => 1
      | n => 2 * exp (n-1)
```

How long does it take to run `exp n`?

- The zero case takes 2 steps:

```
    exp 0
|-> case 0 of 0 => 1 | _ => 2 * (exp (0 - 1))
|-> 1
```

- The $n+1$ case takes 4 steps plus the time for the recursive call:

```
    exp (k+1)
|-> case (k+1) of 0 => 1 | _ => 2 * exp ((k+1)-1)
|-> 2 * exp ((k+1)-1)
|-> 2 * exp k
... recursive call computes a numeral m, so the program is ...
    2 * m
|-> <one more step to do the multiplication>
```

Recurrence. We summarize this by defining a *recurrence relation* $W_{\texttt{exp}}(n)$ representing the number of steps it takes to run `exp` on the numeral n:

$$W_{\texttt{exp}}(0) = 2$$
$$W_{\texttt{exp}}(n) = 4 + W_{\texttt{exp}}(n-1) \text{for non-zero } n$$

Closed form. We can prove the closed form of this recurrence:

Theorem 2. *For all n, $W_{exp}(n) = 4n + 2$.*

In this case, the closed form of $W_{\texttt{exp}}$ is $4n + 2$. Why? Well, you can see that $W_{\texttt{exp}}$ adds 4 for every 1 that it peels off, with a base case of 2. Formally, we can prove this using induction:

Proof. **Case for 0:** To show: $W_{\texttt{exp}}(0) = 4 * 0 + 2$. True by calculation.

Case for $1 + k$: Inductive hypothesis: $W_{\texttt{exp}}(k) = 4 * k + 2$.
To show: $W_{\texttt{exp}}(1 + k) = 4 * (1 + k) + 2$. Proof: By definition, $W_{\texttt{exp}}(1 + k) = 4 + W_{\texttt{exp}}(k)$. By the IH, this equals $4 + 4 * k + 2$ which equals $4 * (1 + k) + 2$. □

The closed form is a better representation for comparison, because it abstracts away from the particular recursion pattern used to define the function.

Big-O $W_{\texttt{exp}}(n)$, which equals $4n + 2$, is $O(n)$. We say that `exp` runs in *linear time*.

4.2 Fast Exponentiation

It turns out that there is an $O(\log n)$ algorithm for exponentiation: divide the exponent by 2 and square the result (assuming multiplication is constant time, which it is for fixed-size integers). That is, we exploit the fact that

$$2^n = (2^{(n/2)})^2$$

Because we're working with integers, we apply this identity only when n is even. When it's odd, we decrement by one.

Define `fun square(x) = x * x`, and note that there is a built-in operation `div` used like `6 div 2`. When `n` is divisible by `k`, `n div k` computes to n/k. We can also implement `evenP` in constant time using the built-in `mod` operator. Then we can define fast exponentiation as follows:

```
fun fexp (n : int) : int =
    case n of
        0 => 1
      | n => (case evenP n of
                  true => square (fexp (n div 2))
                | false => 2 * (fexp (n-1)))
```

It is very important to observe that we are doing something different here: we are making a recursive call on a number other than $n - 1$. This is an example of something called *well-founded recursion*, where you can make recursive calls on *any smaller number*, rather than just the immediate predecessor. Here's the template:

```
fun f (n : int) : T =
... recursive calls on any m < n ...
```

Let's extract a recurrence for the running time. Here's a trick: if you're only going to use a recurrence to compute the O, you don't need to figure out the actual constants, since they won't matter in the end. We'll write k_0, k_1, ... for constant numbers.

$$\begin{aligned}
&W_{\texttt{fexp}}(0) = k_0 \\
&W_{\texttt{fexp}}(n) = k_1 + W_{\texttt{fexp}}(n/2) \quad \text{if } n \text{ is even} \\
&W_{\texttt{fexp}}(n) = k_2 + W_{\texttt{fexp}}(n-1) \quad \text{if } n \text{ is odd}
\end{aligned}$$

$W_{\texttt{fexp}}(n)$ is $O(\log n)$. I won't prove this, but let's see what happens when n is a power of 2. In that case, you always hit the even case, so

$$\begin{aligned}
W_{\texttt{fexp}}(n) &= k_1 + W_{\texttt{fexp}}(n/2) && (1)\\
&= k_1 + k_1 + W_{\texttt{fexp}}(n/4) && (2)\\
&= k_1 + k_1 + k_1 + W_{\texttt{fexp}}(n/8) && (3)\\
&\vdots && (4)
\end{aligned}$$

Since $\log n$ is exactly the number of times you can divide n by 2 to get 1, the closed form is $k_1 \log n + k_2 + k_0$, which is $O(\log n)$.

5 Addtional Example: FastFib

Name that integer sequence: $1, 1, 3, 5, 8, 13, 21, \ldots$. That's right; it's Fibonacci!

Here's the obvious way to implement it:

```
fun fib (n : int) : int =
    case n of
        0 => 1
      | 1 => 1
      | _ => fib (n - 1) + fib (n - 2)
```

Live `evenP` in lab, this function has *three* cases—zero one and $2 + n$. Like exponentiation, this uses *complete induction*: it makes recursive calls not just on a number that is one-smaller, but on any smaller number. For exponentiation, that meant half; here it means both the immediate predecessor and one before that.

Because of these two recursive calls, the recurrence for the work looks like this:

$$W_{\texttt{fib}}(0) = k_0$$
$$W_{\texttt{fib}}(1) = k_1$$
$$W_{\texttt{fib}}(n) = W_{\texttt{fib}}(n-1) + W_{\texttt{fib}}(n-1) \text{for non-zero } n$$

This is not so helpful, since it says that the time to compute the n^{th} Fibonacci n is ...he n^{th} Fibonnaci number!.

However, if we can get an upper bound for this recurrence as follows:

$$W_{\texttt{fib}}(0) = k_0$$
$$W_{\texttt{fib}}(1) = k_1$$
$$W_{\texttt{fib}}(n) \leq 2W_{\texttt{fib}}(n-1) \text{for non-zero } n$$

because $W_{\texttt{fib}}(n)$ is *monotonically increasing* (it's never smaller on bigger inputs).

If you write it out, you can see that the closed form of this recurrence is

$$W_{\texttt{fib}}(n) = k_0 + k_1 + k_2 * 2^{n+1} - 1$$

To see this, you can write the recursion out as a tree. `fib` does `k2` work at each recursive call, so we can label each node with `k2`. Each node has two children, because each call makes two recursive calls.

```
      k2
   k2     k2
 k2 k2   k2 k2
...
```

The k_2 is uniform, so factor it out

```
      1          1
   1      1      2
 1   1  1   1    4
...
```

We want to count the number of nodes in this tree. The total has the form $1+2+4+8+16+\ldots$. The reason is that the tree has n levels, because the recurrence recurs on $n-1$, and the i^th level has 2^i work. Thus, the total amount of work is

$$\sum_{i=1}^{n} 2^i$$

If you look it up, the closed form of this sum is $2^{i+1}-1$ (cf. how many binary numbers are there with n bits).

Once you've written out the closed form, it's clear that this recurrence is $O(2^n)$, just by forgetting the constants.

Fastfib How can we do better? We don't really need two recursive calls:

```
To compute     We need
fib n          fib (n-1) and fib (n-2)
fib (n-1)      fib (n-2) and fib (n-3)
fib (n-2)      fib (n-3) and fib (n-4)
```

So we really don't need two recursive calls, if we reuse the same computaton of `fib n` the two times we use it. The key idea is to *generalize* the problem so that we compute both `fib n` and `fib (n-1)`.

You will implement this in lab.