Lecture 15: More Analysis, HOPs, Lambda
10:00 AM, Oct 11, 2017

Contents

1 Analysis review 1

2 Comparing functions, and big-O 1
   2.1 A brief discussion on notation 4
   2.2 A practical consequence 4

3 Higher Order Procedures 5

4 Lambda 5

5 Summary 7

Objectives

By the end of this class, you will be able to:

- Prove a few more things about big-O
- Go from a recurrence relation for the operation-counting function (for at least some programs) to a big-O estimate of the runtime.

1 Analysis review

In analyzing the runtime performance (i.e., the number of elementary operations used in performing a procedure on an input of some size) we’ve gone through several steps, and have a few left to do.

1. Name and describe the operation-counting procedure (“Let $H(n)$ be the largest number of elementary operations performed by ... ”).

2. Write a recurrence relation for the operation-counting function by looking carefully at the code.

3. Use plug-n-chug to work out a few values for the op-counting functions (e.g., $H(0), H(1), H(2), H(3)$); typically we don’t actually find values for these, but upper bounds, writing things like $H(1) \leq 2A + B$).

4. Make a conjecture about a general pattern for these expressions, leading to a “closed form” expression for $H$ (i.e., one where the formula for $H(n)$ makes no mention of $H(n-1)$ or any other prior values).
5. Prove that this conjecture is correct. [We have not yet done this step.]

6. Say what big-O class the function must belong to [We’ll start this today].

2 Comparing functions, and big-O

We’re going to hold off on proving the conjecture is correct, and instead talk about the last step. To do so, I have to say what a big-\(O\) class is.

To do so, I have to go back to the things you learned on the Homeworks: the notion of “\(f\) is eventually less than \(g\)” and “\(f\) is less than \(g\) up to constants”. Notice that both of these are statements about whole functions, which is an extension of the kinds of things you used to do in algebra class, where you mostly talked about numbers, and eventually proved things like \(x < 2y\), i.e., that two numbers had a particular relationship. We’re talking about functions having a particular relationship to each other.

In everything that follows, I’m going to be talking about functions from \(\mathbb{N}\) to \(\mathbb{N}\), because all we care about in this class are operation-counting functions for procedures. The size of the data consumed by the procedure (typically a list) will be a natural number; the number of operations will also be a natural number. So everything, from here on in this discussion of analysis, is an \(\mathbb{N} \to \mathbb{N}\) function.

We’re going to combine these two ideas into one, and say that “\(f\) is eventually less than \(g\), up to constants”, if there are numbers \(M, c > 0\) with the property that for every integer \(n > M\), we have

\[
f(n) \leq c \cdot g(n).
\]

(Again: this definition applies only to \(\mathbb{N} \to \mathbb{N}\) functions!)

Thinking of \(f\) and \(g\) as op-counting functions for a couple of procedures that produce the same result (e.g., two different ways of implementing “set-intersection”), this dominance notion says that \(f\)’s procedure may not be as good as \(g\)’s for small problems (\(n \leq M\)), but that once the problems are large enough (\(n > M\)), \(f\) performs at least as well as \(g\), at least if we’re willing to ignore a constant multiple.

If we write (for the next few minutes), \(f << g\) to mean “\(f\) is eventually less than \(g\), up to constants”, then there are some basic facts that are pretty useful.

- For any function \(f\), we have \(f << f\).
- If \(f << g\) and \(g << h\), then \(f << h\).
- If \(f(n) = 0\) for every \(n\), then \(f << g\) for every function \(g\).

The last of these is easy to prove: you pick \(M = 0\) and \(c = 1\).

That’s a general form for these proofs: to show that there are numbers \(M\) and \(c\) with the desired property, you say what they are, and then prove that those numbers have the desired property.

I’ve made some claims here, but haven’t proved them, because those proofs are easier once you’ve seen a couple of very concrete examples.
Let’s do a first example. I claim that the function $f(n) = 3n$ is eventually less than the function $g(n) = n$, up to constants.

To prove this, I have to show that they satisfy the definition, so I have to exhibit numbers $M$ and $c$ with the required property.

To find those numbers, I do a little background work: I draw the graphs (use Desmos!) and see that $g$ looks as if it’s mostly less than $f$. But if I multiply $g$ by some constant, I can make it just as big as $f$ . . . and a good constant for doing that is 3. Once I do so, I also have to pick a value $M$ whose meaning is “to the right of this value, I can see that my multiple of $g$ is at least as big as $f$.” Well, since $f = 3g$, just about any value of $M$ will work, and I’m going to choose one, namely $M = 1$, to make things concrete.

OK. Now I’m ready to go.

I claim that $f$ is eventually less than $g$ up to constants, with $M = 1$ and $c = 3$ as proof of that.

I can restate this claim:

Claim: Picking $M = 1$ and $c = 3$, for $n > M$, $f(n) \leq cg(n)$.

I can restate that again, simplifying a little bit:

Claim: for $n > 1$, $f(n) \leq 3 \cdot g(n)$.

Is this last claim true? Well, for any $n > 1$ (indeed, for any $n$ at all), we have

$$f(n) = 3n$$

by the definition of $f$. And we have

$$3 \cdot g(n) = 3 \cdot n$$

by the definition of $g$. So

$$f(n) = 3n \leq 3 \cdot n = 3 \cdot g(n),$$

and we’ve proved our claim.

Let’s do that again for a slightly more interesting case, where “eventually” will come into play.

Let’s look at $f(n) = 3n + 5$ and $g(n) = n$. Once again, graphing shows that not only is $f$ always bigger than $g$, by no matter how much you multiply $g$ by, $f(0)$ will always be bigger than $c \cdot g(0)$. So we’ll definitely need to consider the “eventually” part of things. If, as above, we multiply $g$ by 3, we’ll be comparing $3n + 5$ to $3n$, and the $3n$ will always be less. But who says we have to use 3?

Let’s pick $c = 4$, because graphing shows that when we do that, $3n + 5$ is eventually less than $4n$.

Let’s continue with our preparation work by asking “How big does $n$ have to be for $3n + 5$ to be less than or equal to $4n$?” Well, if we solve

$$3n + 5 \leq 4n$$

$$5 \leq 4n - 3n$$

$$5 \leq n$$

it appears that $n > 4$ is sufficient. So we’ll pick $M = 4$ and $c = 4$. (In today’s slides, I picked $M = 5$; both work fine!)
Question: Why can you pick $M = 4$ when $n$ must be greater than 5?

Answer: Recall that $n > M$, and since we are working with natural numbers, if $n > 4$ then $n \geq 5$.

So my claim is this:

Claim: for $M = 4, c = 4$, we have that for $n > M, f(n) \leq c \cdot g(n)$.

We restate this by plugging in the values of $M$ and $c$ to get

Claim: $n > 4, f(n) \leq 4 \cdot g(n)$.

Is this true? Well, suppose $n > 4$. Adding $3n$ to both sides, we get $3n + n > 3n + 4$, i.e. $4n > 3n + 4$. Now that means that $4n \geq 3n + 5$, which is the same as $3n + 5 \leq 4n$, which is the same as $f(n) \leq 4g(n)$, and we’re done.

This idea — one function eventually being no more than another, up to constants, is so special that it gets its own name and notation.

We define the set $O(g)$ to be all functions $h$ with the property that $h \ll g$, i.e., all functions that are eventually less than $g$.

So we’ve shown, above, that

\[
(n \mapsto 3n) \in O(n \mapsto n) \\
(n \mapsto 3n + 5) \in O(n \mapsto n)
\]

2.1 A brief discussion on notation

You may have heard or seen the expression $O(n)$ before. However, this notation is misleading, as $n$ is not a function. In this class, we write $O(n \mapsto n)$ to show that we are talking about a set of functions.

I’m going to write down two functions, and I want you to think about whether they are the same function or different. You might want to review the definition of “sameness” for functions before committing yourself to an answer.

\[
h : \mathbb{N} \to \mathbb{N} : n \mapsto n + 1 \\
k : \mathbb{N} \to \mathbb{N} : k \mapsto k + 1
\]

So . . . are $h$ and $k$ the same function?

They are, because they have the same domain, and the same codomain, and for every element $x$ in the domain, the value $h(x)$ is the same as the value $k(x)$.

That means that the sets $O(n \mapsto n)$ and $O(k \mapsto k)$ denote exactly the same thing. By long tradition, computer scientists use the letter $n$, but the letter $k$ (or any other) is equally good.

2.2 A practical consequence

We already showed (except for a missing proof of the conjecture) that the operation-counting function, $L$, for the length procedure satisfied a particular recurrence, and that in fact there are
nonnegative numbers $A$ and $B$ with $L(n) \leq Bn + A$ for all $n \in \mathbb{N}$.

Using a proof like the one above (picking $c = B + A$, for instance, and $M = 1$), we can show that $L$ is eventually less than (up to constants) the function $n \mapsto n$, hence that $L \in O(n \mapsto n)$. Informally, we say that “length is a $O(n \mapsto n)$ procedure.” It’s important to remember that what this really means is that the op-counting function for length is in the class $O(n \mapsto n)$.

Let’s dignify that example with a theorem that we can apply to lots of other examples as well.

**Theorem 1.** For any $A, B \geq 0$, the function $f(n) = An + B$ is in $O(k \mapsto k)$ (or, equivalently, $O(n \mapsto n)$).

To see the proof for this theorem, check out the slides.

We went over one other theorem:

**Theorem 2.** For any $A, B, C \geq 0$, if $f$ satisfies a recurrence of the form:

\[
\begin{aligned}
f(0) &= C \\
f(n) &\leq An + B + f(n - 1)
\end{aligned}
\]

for $n > 0$, then $f \in O(n \mapsto n^2)$.

## 3 Higher Order Procedures

Let’s consider a function that improves a list of integers by changing every element of the list to 17.

```Scheme
;;; improve: (int list) -> (int list) 
;;; Input: a list of integers, aloi 
;;; Output: a list of integers such that all integers 
;;; in aloi have been changed to be the number 17 
;;; (define (improve aloi)
  (cond
    [(empty? aloi) empty]
    [(cons? aloi) (cons
      17
      (improve (rest aloi)))]))
```

We’ve seen functions of this general format a lot: functions that do *something* to every element of a list. It turns out that this type of function is so common, that there is a much shorter way to write them:

```Scheme
(map proc list)
```

Where `proc` is the procedure to apply to every element of the list, and `list` is the list that you want to change.

The type signature for map is something new - this is the first time that we’ve used a procedure as input. It looks like this:
; map: ('a -> 'b)* ('a list) -> ('b list)
In words, this says that the arguments to map are: 1) a procedure with input of 'a data and output of 'b data, and 2) a list of 'a data (the same data type as the input to the procedure).
The output of map is a 'b list (the same data type as the output of the procedure.)
Map is known as a **higher order procedure** (HOP) because one of its arguments is a function.

## 4 Lambda

We now introduce a new bit of syntax in Racket: **lambda**. A lambda-expression evaluates to a closure. That is to say, informally, lambda-expressions are a way to produce procedures without using a **define**.

The syntax of a lambda-expression is:

```
(lambda name-list body)
```

When evaluated, this produces a user-defined procedure (i.e., a closure), with the name-list as the argument list, and body as the body.

Here’s a typical one:

```
(lambda (x) (+ x 1))
```

If we wanted to apply our user-defined procedure to a variety of integers to see what it would output, we’d do something like,

```
((lambda (x) (+ x 1)) 5)
=> 6
((lambda (x) (+ x 1)) 17)
=> 18
```

This produces a closure whose argument-list is x and whose body is (+ x 1).

More generally, a lambda-expression may have many arguments and any expression at all as its body, so you can write

```
(lambda (x y) (* (+ x y) (- x 2)))
```

for instance.

Recall the **improve** function from the last section. Now, using **map** and **lambda**, we can rewrite the procedure like so:

```
(define (improve aloi)
  (map (lambda (x) 17) aloi))
```

To break this down: using **lambda**, we’re creating a procedure that for any input x, the output is 17. Then, we’re applying that function to every element of aloi. So, the output of **improve** is aloi, with every element replaced with 17.

And now the secret from earlier in the semester:
(define (f x) (+ x 1))

is really just syntactic sugar for

(define f (lambda (x) (+ x 1)))

The things produced by lambda-expressions are called “anonymous” because they don’t have names. They might seem useless now, but soon they’ll be your best friends.

5 Summary

Ideas

- We’ve proven that any procedure whose operation counting function takes the closed form $f(n) = An + B$ for any $A, B \geq 0$ is in $O(n \mapsto n)$ (Theorem 1).

- We’ve proven that any procedure whose operation counting function has a recurrence relation which takes the form

$$
\begin{align*}
    f(0) &= C \\
    f(n) &\leq An + B + f(n - 1), \text{ for } n > 0
\end{align*}
$$

for any $A, B, C \geq 0$ is in $O(n \mapsto n^2)$ (Theorem 2).

- All big-O run time proofs in CS 17 follow very similar formats. If you get stuck, it’s always safe to refer back to the proofs provided in lecture notes, and use them as a guide! Most often, you’ll find that what differs most from proof to proof is the actual algebra involved.

- Higher order procedures (HOPs) are procedures that consume a function, rather than atomic or compound data.

Skills

- We learned the process for determining the big-O class of a recursive procedure. Namely, we learned how to develop a recurrence relation (which tells us something about the worst-case-scenario run time of our procedure in the base case and the recursive case) and use “plug-n-chug” to turn that recurrence relation into an equation. Once we have our equation, we know to use a proof-by-contradiction using the well-ordering principle to establish that our equation holds true for all $n$ greater than some natural number, and therefore must be in some big-O class.

- We can now used `lambda` to create anonymous procedures.
Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)