Lecture 11: Two-Argument Recursion

10:00 AM, Sep 28, 2018

Contents

1 Full Notes ......................................................... 1
   1.1 Worst-case analysis ...................................... 1
   1.2 Recurrence inequalities ................................... 3
   1.3 my-reverse .................................................. 4

2 Summary ......................................................... 6
   2.1 Worst-case analysis ...................................... 6
   2.2 Recurrence inequalities ................................... 7
      2.2.1 Linear-time ........................................... 7
   2.3 Helpers .................................................... 8

1 Full Notes

Last time we saw how to derive a recurrence relation (equality) for a single recursive procedure. Today’s topics:

- The Devil Cat (i.e. worst-case analysis)
- Recurrence relations other than equality relations.
- Not specifying the constants.
- The Big O.

1.1 Worst-case analysis

double-like

- **input**: list $L$
- **output**: list obtained from $L$ by replacing each occurrence of the symbol `like` with two occurrences
(define double-like
  (lambda (L)
    (cond
      ((empty? L) empty)
      ((equal? (quote like) (car L))
       (cons (quote like) (cons (quote like) (double-like (cdr L))))
      (#true (cons (car L) (double-like (cdr L))))))))

> (double-like '(And I was like duh and he was like no and I am so like done))
(And I was like like duh and he was like like no and I am so like like done)

Can you tell I have two teenage children?

Let’s try to formulate a recurrence relation.

“We define the function $g$ as follows. Let $g(n)$ be the number of operations executed by double-like when given an input of size $n$.”

- case of empty list: 6 operations
- case of list starting with like: $16 + f(n-1)$ operations
- case of list starting with something other than like: $15 + f(n-1)$ operations.

The case in which $L$ is empty works out fine. We get $g(0) = 6$. When $L$ is not empty ($n > 0$), the number of operations is 15 or 16 (not including the recursive call), depending on whether the first item is like. This seems bad. Take the case $n = 1$. Is $g(1)$ supposed to be 15 or 16? It cannot be both, and the value of $g(n)$ can depend only on the size $n$ of the input, not on the specifics of the input.

We clearly have to change the definition of $f$, but how?

The traditional approach (which is usually what you want) is worst-case analysis. Let’s say you’re writing some software. You want to provide a performance guarantee. You want to say: this procedure always runs in linear time or whatever, regardless of the particular input.

For this reason, we revise our incantation:

“We define the function $g$ as follows. Let $g(n)$ be the maximum number of operations executed by double-like when given any input of size $n$.”

That is, even if with the worst possible size-$n$ input, the time is only $g(n)$.

You’re supposed to imagine that the devil, who knows your code and is very clever, is choosing the inputs. I used to try to draw the devil, and it turned out looking more like a cat, so we decided it was the devil cat. Anyway, the devil cat knows your code, and chooses inputs to make it slow.

This does not mean that as part of your analysis you should identify which input is the worst! Never do that. You’ll probably get it wrong.
When doing worst-case analysis, we often do not aim for a recurrence equality. Instead, we aim for a recurrence inequality.

### 1.2 Recurrence inequalities

Returning to the analysis of `double-like`:

- case of empty list: 6 operations
- case of list starting with `like`: 16 + g(n − 1) operations
- case of list starting with something other than `like`: 15 + g(n − 1) operations.

We cannot write \( g(n) = 15 + g(n − 1) \) because that would be a lie. We also cannot write \( g(n) = 16 + g(n − 1) \).

However, it’s true in either case that \( g(n) \leq 16 + g(n − 1) \), so that’s what we write down:

\[
\begin{align*}
g(0) &= 6 \\
g(n) &\leq 16 + g(n − 1) \text{ for } n > 0
\end{align*}
\]

We previously had the theorem:

| **Theorem:** For any numbers \( a \) and \( b \), if \( f \) is a function such that 
| \[
\begin{align*}
f(0) &= a \\
f(n) &= b + f(n − 1) \text{ for } n > 0
\end{align*}
\] then \( f(n) = a + bn \).

As a consequence, we have a corollary:

| **Corollary:** For any numbers \( a \) and \( b \), if \( g \) is a function such that 
| \[
\begin{align*}
g(0) &\leq a \\
g(n) &\leq b + g(n − 1) \text{ for } n > 0
\end{align*}
\] then \( g(n) \leq a + bn \).

Using the corollary, we can conclude that the function \( g(\cdot) \) we defined for the number of operations of `double-like` satisfies

\[
g(n) \leq 6 + 16n
\]

Based on this, we say that the procedure is a linear-time procedure.

You could also show that \( g(n) \geq 6 + 15n \).

We say a procedure is a **linear-time procedure** if there are numbers \( a \) and \( b \) such that the maximum number of operations performed by the procedure on an input of size \( n \) is at most \( a + bn \).
Similarly, a *quadratic-time* procedure is one such that there are constants $a, b, c$ such that the worst-case number of operations performed by the procedure on an input of size $n$ is at most $a + bn + cn^2$.

A *constant-time* procedure is one such that there is a constant $a$ such that the worst-case number of operations is at most $a$.

According to the formal definitions I have given so far, a constant-time procedure qualifies as a linear-time procedure and a linear-time procedure qualifies as a quadratic-time procedure, but it would be misleading to say, for example, that `double-like` is a quadratic-time procedure—technically correct, but if you heard that, you would assume it was not a linear-time procedure, that quadratic-time procedure is the best characterization.

Next time we will talk about Big O and Big Omega, which are more general concepts that capture all this.

For today, the important thing is this: in order to show that a procedure is a constant-time procedure, **you don’t have to know exactly how many operations it does.** You just have to know that there are constants $a$ and $b$ such that

- for inputs of size zero, at most $a$ operations are performed, and
- for inputs of size $n > 0$, at most $b$ operations are performed other than the recursive call, and the input to the recursive call is of size $n - 1$.

This means you don’t have to actually count the operations. You just have to make sure that, other than the recursive call, all other computations require a number of operations that does not depend on the size of the input.

What about if if the input to the recursive call is $n - 2$ instead of $n - 1$? What if it could be $n - 1$ or $n - 2$ depending on the input?

We use a slightly revised definition and a slightly different theorem.

“Let $g(n)$ be the maximum number of operations performed by procedure X on inputs of size at most $n$.”

Revised theorem:

<table>
<thead>
<tr>
<th><strong>Theorem:</strong> For any integers $a$ and $b$, if $g$ is a function such that</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g(0) \leq a$</td>
</tr>
<tr>
<td>$g(n) \leq b + f(m)$ where $m &lt; n$, for $n &gt; 0$</td>
</tr>
<tr>
<td>then $g(n) \leq a + bn$.</td>
</tr>
</tbody>
</table>

### 1.3 my-reverse

Let’s look at some procedures you wrote in lab.

We asked that you write `my-reverse` in terms of the procedures `last` and `all-but-last`.

Ordinarily, things might proceed in the other order: you set out to write a list-reversing procedure, and you realize that it would be easy if you only had `last` and `all-but-last`. You would then
refer to last and all-but-last as helper procedures for my-reverse.

Figuring out which helper procedures you want to use requires some creativity and problem-solving. Don’t expect to necessarily figure it out right off the bat. It might take several tries, even once you’re more experienced.

- Sometimes you can identify some natural subtask whose solution helps you with the task at hand; that’s the case with my-reverse.
- Sometimes the task is not so natural; often it involves an additional argument or two to the original task.
- There’s another use that just involves using a formal argument of the helper procedure to hold onto the value of some expression so you don’t have to compute it twice. (There is another way to achieve this, using a special form we haven’t yet taught.)

```
(define my-reverse
  (lambda (L)
    (cond ...
      (#true ... (last L) ... (my-reverse (all-but-last L))))...)
```

Let’s assume you’ve already analyzed last and all-but-last, and found that they are linear-time procedures. That means there are constants \(a_1, b_1, a_2, b_2\) such that

- The worst-case number of operations performed by last on inputs of size \(n\) is at most \(a_1 + b_1 n\).
- The worst-case number of operations performed by all-but-last on inputs of size \(n\) is at most \(a_2 + b_2 n\).

Let’s analyze my-reverse.

We define the function \(f\) as follows: Let \(f(n)\) be the worst-case number of operations performed by my-reverse on inputs of size at most \(n\).

Consider the application of my-reverse to list \(L\). If \(L\) is empty, the number of operations is some constant \(d\), so \(f(0) = d\).

Otherwise, the operations performed include

- operations performed by a call to last on \(L\): at most \(a_1 + b_1 n\)
- operations performed by a call to all-but-last on \(L\): at most \(a_2 + b + 2n\)
- a recursive call on a list of size one less than that of \(L\).
- some constant \(c\) number of other operations

Totalling that up. Let \(a = a_1 + a_2 + c\) and let \(b = b_1 + b_2\).

\[
\begin{align*}
f(n) & \leq a_1 + b_1 n + a_2 + b_2 n + c + f(n-1) \\
& \leq (a_1 + a_2 + c) + (b_1 + b_2)n + f(n-1) \\
& \leq a + bn + f(n-1)
\end{align*}
\]
How to analyze this? Example:

\[
\begin{align*}
    f(4) & \leq a + b \cdot 4 + f(3) \\
    & \leq a + b \cdot 4 + a + b \cdot 3 + f(2) \\
    & \leq a + b \cdot 4 + a + b \cdot 3 + a + b \cdot 2 + f(1) \\
    & \leq a + b \cdot 4 + a + b \cdot 3 + a + b \cdot 2 + a + b \cdot 1 + f(0) \\
    & \leq a + b \cdot 4 + a + b \cdot 3 + a + b \cdot 2 + a + b \cdot 1 + d \\
    & \leq (a + a + a + a) + (b \cdot 4 + b \cdot 4 + b \cdot 4 + b \cdot 4) + d \\
    & \leq 4a + 4b \cdot 4 + d
\end{align*}
\]

Similarly,

\[
\begin{align*}
    f(n) & \leq a + b \cdot n + f(n - 1) \\
    & \leq a + b \cdot n + a + b \cdot (n - 1) + f(n - 2) \\
    & \leq a + b \cdot n + a + b \cdot (n - 1) + a + b \cdot (n - 2) + f(n - 3) \\
    & \leq a + b \cdot n + a + b \cdot (n - 1) + a + b \cdot (n - 2) + a + b \cdot (n - 3) + f(n - 4) \\
    & \vdots \\
    & \leq a + b \cdot n + a + b \cdot (n - 1) + a + b \cdot (n - 2) + a + b \cdot (n - 3) + \cdots + a + b \cdot 1 + f(0) \\
    & \leq a + b \cdot n + a + b \cdot (n - 1) + a + b \cdot (n - 2) + a + b \cdot (n - 3) + \cdots + a + b \cdot 1 + d \\
    & \leq (a + a + \cdots + a + b \cdot n + b \cdot n + \cdots + b \cdot n) + d \\
    & \leq an + bn^2 + d
\end{align*}
\]

This shows that the procedure is a quadratic-time procedure.

It turns out that there is a linear-time procedure for reversing a list. We’ll learn that soon.

## 2 Summary

### 2.1 Worst-case analysis

Let’s look at the following procedure, `double-like` which takes as input a list, \( L \), and output the list obtained from \( L \) by replacing each occurrence of the symbol `like` with two occurrences:

```scheme
(define double-like
  (lambda (L)
    (cond
      ((empty? L) empty)
      ((equal? (quote like) (car L)) (cons (quote like) (cons (quote like) (double-like (cdr L))))))
      (#true (cons (car L) (double-like (cdr L)))))))
```

6
Now, we define $g(n)$ as the maximum number of operations executed by double-like when given an input of size $n$. Let’s examine the following cases:

- case of empty list: 6 operations
- case of list starting with like: $16 + f(n - 1)$ operations
- case of list starting with something else: $15 + f(n - 1)$ operations.

### 2.2 Recurrence inequalities

We know the following theorem from our analysis of double-like:

For any real numbers $a$ and $b$, if $f$ is a function such that

\[
\begin{align*}
  f(0) &= 6 \\
  f(n) &= b + f(m) \text{ where } m < n, \text{ for } n > 0
\end{align*}
\]

then $f(n) = a + bn$

This means that even if the worst possible size-$n$ is our input, the time is only $g(n)$.

So, we also know the following corollary:

For any numbers $a$ and $b$, if $g$ is a function such that

\[
\begin{align*}
  g(0) &\leq a \\
  g(n) &\leq b + g(n - 1) \text{ for } n > 0
\end{align*}
\]

then $g(n) \leq a + bn$

So, we know based on the corollary that the following is true (in reference to double-like):

\[
g(n) \leq 6 + 16n
\]

### 2.2.1 Linear-time

- We really want linear-time procedures if we can get them. But, the definition of linear-time procedures is generous, so we really don’t need to count operations very carefully. We just need to be able to tell that the number of operations is less than some constant.

Let’s look at this in terms of my-reverse that we wrote in lab, which calls on two helper procedures:

```scheme
(define my-reverse
  (lambda (L)
    (cond
      ... (#true ... (last L) ...)
      (my-reverse (all-but-last L)))))) ... )
```
Here, we have:
\[ f(n) \leq a + bn + f(n - 1) \]

The following is an example of how to analyze this:

\[ f(4) \leq 4a + 4b \cdot 4 + d \]

Similarly,
\[ f(n) \leq an + bn^2 + d \]

The above shows that the procedure is a quadratic-time procedure.

### 2.3 Helpers

There are several reasons to use helper procedures:

- You have found a natural sub-task, maybe one involving an additional parameter
- To hold the result of a computation you do

(Coming up with the right helper procedure can be challenging!)

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)