Class 12: Analysis

• Warmup: analyze reverse-with-tail
• Generic types
• Plug-n-chug method for guessing a closed-form for recurrences
• Warning
Final version of reverse-with-tail (for generic list)

;reverse-with-tail: ('a list) * ('a list) -> ('a list)
(define (reverse-with-tail alod tail)
  (cond
    [(empty? alod) tail]
    [(cons? alod) (reverse-with-tail (rest alod)
        (cons (first alod) tail))]]))

"Let $H(n)$ be the number of operations in evaluating (reverse-with-tail alod tail) on any input where alod is a list of length $n$.

\[ H(0) = A \]
\[ H(n) = B + H(n - 1) \text{ for } n > 0 \]
What about “reverse” itself?

(define (reverse alod)
    (reverse-with-tail alod empty))

"Let $R(n)$ be the number of operations in evaluating (reverse alod) on any input where alod is a list of length $n$."

Then

$$R(n) = C + H(n)$$

• In other words: once we figure out $H(n)$, $R(n)$ is really easy!
• For non-recursive functions like this, analysis is often similarly simple
Helper procedures

• In writing “reverse”, we used “reverse-with-tail” to get the job done
  • Just as we used “+” to make “length” work

• If you write your own procedures (like reverse-with-tail) that help you solve a problem, they’re called “helper procedures”
  • Still need design-recipe steps
    • Esp. since the helper is often the part doing all the work

• Choosing good helpers takes practice, alas
We keep seeing the same recurrence relation!

\[
\begin{align*}
Q(0) &= A \\
Q(n) &= B + Q(n - 1) \text{ for } n > 0
\end{align*}
\]

We saw something like this for \( L \), the op-counter for length

We saw something like this for \( C \), the op-counter for contains17?

• It had a \( \leq \) instead of =

We saw something like this for \( H \), the op-counter for reverse-helper

It's time to "solve the recurrence"
Solving a recurrence

\[
Q(0) = A \\
Q(n) = B + Q(n - 1) \text{ for } n > 0
\]

• A "solution" is a formula for \(Q(n)\) that does not involve \(Q(n - 1)\) (or any other previous values), i.e., a non-recursive expression for \(Q(n)\).

• In special cases (all we've seen so far), we can actually write
  \[Q(n) = ...\]

• More often, we're at most able to say
  \[Q(n) \leq ...\]
Plug-n-chug

\[ Q(0) = A \]
\[ Q(n) = B + Q(n - 1) \text{ for } n > 0 \]

Step 1: Use recurrence to write out what you know about \( Q(0), Q(1), \ldots \) -- perhaps 3 or 4 steps.

\[
\begin{align*}
Q(0) &= A \\
Q(1) &= B + Q(0) = B + A \\
Q(2) &= B + Q(1) = B + (B + A) = 2B + A \\
Q(3) &= B + Q(2) = B + (B + (B + A)) = 3B + A
\end{align*}
\]
Plug-n-chug

Step 2: Try to notice a *pattern*. Very tough with numbers; often easy with letters.

\[ Q(0) = A \]
\[ Q(1) = B + Q(0) = B + A \]
\[ Q(2) = B + Q(1) = B + (B + A) = 2B + A \]
\[ Q(3) = B + Q(2) = B + (B + (B + A)) = 3B + A \]

\[ Q(n) = nB + A \]
Plug-n-chug

This pattern is called a "closed-form" expression, because there's no recursion.

\[ Q(n) = nB + A \]

It may seem obvious (in this case) that this formula works for every \( n \), but often it's not so obvious. So

Step 3: *Prove that your conjectured solution is actually a solution.* (Next class).
Let’s try that again with numbers

• Suppose the recurrence was
  \[ Q(0) = 9 \]
  \[ Q(n) = 13 + Q(n - 1) \text{ for } n > 0 \]

Working out the first few
\[ Q(0) = 9 \]
\[ Q(1) = 13 + Q(0) = 13 + 9 = 22 \]
\[ Q(2) = 13 + Q(1) = 13 + 22 = 35 \]
\[ Q(3) = 13 + Q(2) = 13 + 35 = 48 \]

It’s much harder to see the pattern in 9, 22, 35, 48, ...
Summary so far

• Each of the op-counting functions we’ve seen...
  • The op-counter for the length procedure
  • The op-counter for contains17?
  • The op-counter for reverse-with-tail

• ...has a recurrence whose solution has the form \( Q(n) = nB + A \)

• In other words, aside from a small constant amount of work \( A \), the work done by the procedure grows in proportion to the length of the input

• We call these procedures “linear time” procedures

• Broadly considered a “good” result in CS
Plug-n-chug practice (I)

\[ Q(0) = A \]
\[ Q(n) = Q(n - 1) \text{ for } n > 0 \]

• By hand write out \( Q(0), Q(1), Q(2), Q(3) \)

• Try to notice a pattern
Plug-n-chug practice (II)

\[ Q(0) = A \]
\[ Q(n) = 2Q(n - 1) \text{ for } n > 0 \]
Plug-n-chug practice (III)

\[ Q(1) = A \]
\[ Q(n) = 2Q(n/2) \text{ for } n \geq 1 \]

(apply only to \( n = 2^k \), a power of 2.)

• Find \( Q(2^0), Q(2^1), Q(2^2), Q(2^3), \ldots \)

• Try to guess a pattern for \( Q(2^k) \) (in terms of \( k \)).
A frequent example

\[ Q(0) = A \]
\[ Q(n) = B + Cn + Q(n - 1) \text{ for } n > 0 \]

The \( Cn \) term might appear if, in the non-recursive-call part, you computed the length of the input, or used \textit{contains17?} on it

\[
\begin{align*}
Q(0) &= A \\
Q(1) &= B + C \cdot 1 + Q(0) = B + C \cdot 1 + A \\
Q(2) &= B + C \cdot 2 + Q(1) = B + C \cdot 2 + B + C \cdot 1 + A = 2B + (1 + 2)C + A \\
Q(3) &= B + C \cdot 3 + Q(2) = B + C \cdot 3 + B + C \cdot 2 + B + C \cdot 1 + A = 3B + (1 + 2 + 3)C + A
\end{align*}
\]

\[ Q(n) = nB + (1 + 2 + 3 + \ldots + n)C + A \]
Gauss's formula for $1 + 2 + ... + n$

$$S = 1 + 2 + \cdots + (n - 1) + n$$
$$2S = 1 + 2 + \cdots + (n - 1) + n + (1 + 2 + \cdots + (n - 1) + n)$$
$$2S = n + (n - 1) + \cdots + 2 + 1 + (1 + 2 + \cdots + (n - 1) + n)$$
$$2S = n + (n - 1) + \cdots + 2 + 1 + (1 + 2 + \cdots + (n - 1) + n)$$
$$2S = (n + 1) + (n - 1 + 2) + \cdots + (2 + n - 1) + (1 + n)$$
$$2S = n(n + 1)$$
$$S = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$
An oft-occurring example, summary

\[ Q(0) = A \]
\[ Q(n) = B + Cn + Q(n - 1) \text{ for } n > 0 \]
\[ Q(n) = nB + (1 + 2 + 3 + \ldots + n)C + A \]

\[ = Bn + C \left( \frac{n^2 + n}{2} \right) + A \]
\[ = C \left( \frac{n^2 + n}{2} \right) + Bn + A \]
\[ = \left( \frac{C}{2} \right) n^2 + \left( B + \frac{C}{2} \right) n + A \]

For large values of \( n \), it’ll turn out that only that first term matters.
We say procedures whose op-count functions are like this are “quadratic time” procedures
Brief review of standard process henceforth

• Write a (recursive) program
• Write down a recurrence relation that it satisfies
• Solution version 1
  • Use plug-and-chug to guess a solution
  • Prove your guess correct
• Solution version 2
  • Recognize the recurrence as one you’ve seen before
  • Quote the prior analysis result
• Say something about the big-O class of the result
A new procedure

(define (my-apply f a)
  (f a))

(check-expect (my-apply first (list 3)) 3)
(check-expect (my-apply not true) false)

Discuss: What should be the type-signature for my-apply?
; my-apply: ??? * ??? -> ???
A new procedure

(define (my-apply f a)
  (f a))

(check-expect (my-apply first (list 3)) 3)
(check-expect (my-apply not true) false)

Solution:
; my-apply: ('a -> 'b) * 'a -> 'b
Warning

• We’ve reached the part of the course where the "I've programmed before" folks tend to hit the wall
• Really, really, really discuss with your partner before coding
• Draw a recursive diagram. Then another. Then another.
• Sometimes procedures make more than one recursive call!
  • In the recursive diagram, you need to do something to show this, like split things into a left- and right-page recursive call and recursive result
  • Use ideation space to imagine how to combine the two results to get a single overall answer.