Homework 3: Recursion

Due: 10:59 PM, Sep 25, 2019

Contents

1 Odds Only (12 Points) 2

2 Increment All (12 Points) 3

3 Lengthen (10 Points) 3

4 Right-Max (12 Points) 3

5 Function Growth Revisited (9 Points) 4

6 Challenge Problem: Left-Max (Optional) 6

Objectives

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

1. Follow the design recipe
2. Use recursion diagrams
3. Use recursion to filter out a list
4. Use recursion to apply the same procedure to every element of a list
5. Use algebraic principles to prove that one function is eventually greater than another function

How to Hand In

Specific instructions for how to hand in this assignment will be posted on Piazza.

For this (and all) homework assignments, you should hand in answers for all the non-practice questions. For this homework specifically, this entails answering the Odds Only, Increment All, Lengthen, Right-Max, and Function Growth Revisited questions.

To hand in your solutions to these problems, you must store them in appropriately-named files. Each should be named for the corresponding problem. Please submit the following files:

- README.txt
- odds-only.rkt
For this assignment, all files you turn in that contain code must be Racket files, so they must end with extension .rkt. If you are using a departmental linux system, all your solution files should reside in your `/course/cs0170/homeworks/hw03` directory.

For this and every future assignment, you should also have a README.txt file whose first line contains only your Banner ID, and optionally with a message to the person grading explaining anything peculiar about the handin. For example:

**README.txt:**
B01234567
There’s nothing to say except that I’m turning in three code files plus this README the way the instructions say that I should.

**To hand in your solutions to these problems, you must upload them to Gradescope. Do not zip or compress them.**

**Introduction**

One of the fundamental tools of functional programming is recursion. For example, you might need to recur through a list and apply the same procedure to all the elements of the list, or you may wish to filter out elements of the list that you don’t want. In this homework, you’ll practice using recursion to do such tasks.

**Design Recipe**

Whenever we ask you to “write a procedure,” you are expected to carry out each and every step of the design recipe, which will evolve over the next week or two. For convenience, here are the steps of the design recipe:

1. Data definition
2. Examples of the data
3. Type signature
4. Call structure
5. Specification
6. Test cases/examples: recursion diagrams if it’s a recursive procedure
7. Template
8. Code
9. Run your program

For more information, you can reference the Design Recipe post on Piazza, or the Design Recipe Guide on the course website.

Problems

1 Odds Only (12 Points)

We can use builtins that return booleans (these are called predicates) to filter lists by removing elements for which the predicate returns false.

Task: Write a procedure, odds-only, that consumes (i.e., takes as input) a list of ints, aloi, and produces (i.e., outputs) another list of ints that contains only the odd integers in aloi, in their input order. Use the predicate odd? to determine whether the integer is odd or not.

Examples:

\[(\text{odds-only (cons 1 (cons 2 (cons 3 empty)))}) \Rightarrow (\text{cons 1 (cons 3 empty)})\]
\[(\text{odds-only (cons 2 (cons 18 empty)))} \Rightarrow \text{empty}\]

2 Increment All (12 Points)

Task: Write a procedure, increment-all, that takes as input a list of nums, alon, and outputs a new list where each element is the result of adding one to the corresponding number in the old list.

Examples:

\[(\text{increment-all empty}) \Rightarrow \text{empty}\]
\[(\text{increment-all (cons 1 empty)}) \Rightarrow (\text{cons 2 empty})\]
\[(\text{increment-all (cons 1 (cons 2 (cons 5 empty)))}) \Rightarrow (\text{cons 2 (cons 3 (cons 6 empty))})\]
\[(\text{increment-all (cons -1 (cons 5 (cons -3 empty)))}) \Rightarrow (\text{cons 0 (cons 6 (cons -2 empty))})\]

3 Lengthen (10 Points)

Task: Write a program lengthen that consumes a list of nums (alon), and sticks an extra 0 onto the end of it.
Examples:

\[
\text{(lengthen (cons 1 (cons 2 \text{empty}))}) \Rightarrow (\text{cons 1 (cons 2 (cons 0 \text{empty}))})
\]

\[
\text{(lengthen (cons 0 \text{empty}))} \Rightarrow (\text{cons 0 (cons 0 \text{empty})})
\]

Be sure to include recursion diagrams and follow the design recipe.

4 Right-Max (12 Points)

Task: Given a list of ints, aloi, write a program right-max to produce a new list in which each

   element is replaced by the maximum of itself and all the elements following it.

Example:

\[
(\text{cons 5 (cons 1 (cons 3 \text{empty}))}) \Rightarrow (\text{cons 5 (cons 3 (cons 3 \text{empty}))})
\]

Looking at the element 1 in the first list, the largest element in the list starting from 1 is the 3 in

   the next spot, so we replace the 1 by a 3 in the output.

Part 1: Draw a recursion diagram for the input \text{empty}, for \text{(cons 10 \text{empty})}, for \text{(cons 4 (cons 1 (cons 5 \text{empty}))}, and for \text{(cons 6 (cons 1 (cons 5 \text{empty}))}). You can do these

   in the comments before your program. For each one, be sure to fill in the “ideation space” with a
description of how to get from the recursive output to the overall output.

Part 2: Follow the design recipe to write the program. This is a case where writing good test/example
cases will really help you.

You may, if you want to, use, in your program, the Racket procedure max2, defined by:

\[
\text{(define (max2 a b) (if (> a b) a b))}
\]

which consumes two nums and returns the larger one.

(To use this, you’ll need to paste it into the top of your racket program. You do not need to write

   out the design-recipe for this procedure, as it’s not your program. You do not need to use max2 if

   you don’t want to.)

5 Function Growth Revisited (9 Points)

On Homework 2, we compared functions to make statements like “\(f\) is eventually larger than \(g\).”

For this problem, we’re going to do that again. We’ll look at functions from \(\mathbb{N}\) to \(\mathbb{R}_+\).

The focus on this set of functions is motivated by problems in computer science where often we
don’t care about what happens in small cases of a problem, but as the problems we try to solve
grow larger (e.g., computing the length of larger and larger lists) our solution method may take

   more and more steps to complete its work. The number of steps, \(S(n)\), depends on the size of the
problem, \( n \), in a way that we’ll quantify in class. But before we do so, we’d like to be able to talk about two step-counting functions \( S \) and \( T \), and say that “for big problems, \( S \) looks better” or “\( T \) looks better.”

On Homework 2, you checked that one function was larger than another by graphing, and that’s a great place to start. For this problem, we’re going to use algebra.

Recall the definition:

If \( f, g : \mathbb{N} \to \mathbb{R}_+ \) are functions, and there’s some number \( M \) with the property that whenever \( n \geq M \), we have \( f(n) > g(n) \), we say that \( f \) is eventually larger than \( g \) (or “eventually bigger than” or “eventually greater than”, etc.)

Suppose that \( T_1(n) = 2n \) and \( S_1(n) = 8 \). We’ll now explain, with a little algebra, why for \( n > 5 \), we have \( T_1(n) > S_1(n) \). This will establish that \( T_1 \) is eventually greater than \( S_1 \), with \( M = 5 \) providing the proof of this. The following is a model for what you should write for all solutions:

We pick \( M = 5 \).

Suppose that \( n \geq M \). Then we have

\[
\begin{align*}
  n &\geq 5 \quad \text{because } M = 5 \\
  2n &\geq 10 \quad , \text{ by doubling the previous inequality} \\
  2n &> 8 \quad , \text{ because } 10 > 8 \\
  T_1(n) &> S_1(n) \quad , \text{ by substitution of the definitions of } T_1 \text{ and } S_1.
\end{align*}
\]

Thus (picking \( M = 5 \)), we have that \( T_1 \) is eventually larger than \( S_1 \). \footnote{The choice \( M = 4 \) almost works; if our definition of “eventually greater than” had said \( n > M \) rather than \( n \geq M \), it would have. The choice of greater-than vs greater-than-or-equal makes no real difference in the main ideas, but since I wanted to be consistent with HW2, I left it this way.}

This “two-column proof” format, where every step has an associated reason on the right hand side, is a great way to ensure that you really have got a logical proof of the thing you’re trying to prove.

Here are some steps you can do in the algebra that proves an inequality:

- Add a nonnegative number (like “4” or \( n \)) to the big side, so you can change from \( A < B \) to \( A < B + n \). This is because, if \( B \) is greater than \( A \), \( B \) plus a positive number certainly is as well. This doesn’t come up very often, however.

- Add or subtract the same number (which may be positive or negative or zero – no constraints!) to both sides.

- Multiply or divide both sides of the inequality by the same \textit{positive} number. Since \( n \) will always be positive, multiplying both sides by \( n \) or \( n + 2 \) or \( n + 17 \) are all allowed.

The main rule of proofs like this is that you have to start with something that’s true and eventually reach the statement that you need to prove, \textit{not} the other way around. So in the steps above, I started from \( n > 4 \) and eventually reached \( T_1(n) > S_1(n) \).

The dirty little secret is that almost everyone starts from the statement to be proved and works backwards (on a piece of scratch paper) and then rewrites things in a forward direction, providing justifications for each step. So for the example above, I might have written
Hmm. I want $T_1(n) > S_1(n)$, which means $2n > 8$. I can divide by 2 to get $n > 4$. Oh! So for numbers $n$ larger than 4, I should be OK. I’ll pick $M = 5$, because then any $n \geq M$ will be at least 5, and therefore greater than 4.

And then I’d have reversed everything and written out the version above.

There’s a common problem, which is that you find yourself wanting to divide by $n$, but since $n$ is a natural number, it might be zero, and dividing by zero isn’t allowed. On the other hand, suppose you make it a rule for yourself that you’re always going to pick $M$ to be at least 1. Then you’ll know that $n \geq 1 > 0$, so you can divide through by it. So when I pick $M$ values, I always make a mental note: even if the algebra ends up saying that $M = 0$ is OK, I should pick $M = 1$. As I’ve defined things, using greater-than signs rather than greater-than-or-equal signs, it turns out that $M = 0$ often works fine. But other folks use different conventions, and the “pick $M$ to be at least 1” trick turns out to work for all of them. So I use it even when it isn’t necessary.

**Task:** Suppose that $S_2(n) = n$ and $T_2(n) = n^2$. Explain, with a little algebra, why $T_2$ is eventually larger than $S_2$.  
**Hint:** Pick $M = 2$, or any number larger than 2.

**Task:** Suppose that $S_3(n) = 2n + \frac{36}{n}$, and that $T_3(n) = 3n$ for every $n \in \mathbb{N}$. Explain, with a little algebra, why $T_3$ is eventually larger than $S_3$.  
**Hint:** Can you find, by experimentation or graphing, a good choice for $M$, as you did in HW 2? Your answer doesn’t have to be the smallest possible $M$; sometimes picking a slightly larger number as $M$ makes the algebra easier. For this problem, either the smallest possible or a larger value are good starting points.

**Task:** Suppose that $T_4(n) = 3S_4(n)$, and that $S_4(n) > 0$ for every $n \in \mathbb{N}$. Explain, with a little algebra, why $T_4$ is eventually greater than $S_4$.  
**Hint:** It’s enough to show that $T_4(n) > S_4(n)$ for every $n \in \mathbb{N}$; once you do that, picking $M = 1$ in the definition suffices.

**Note:** Format your answers in your .txt file similarly to the short proof written above for $S_1$ and $T_1$.

Given two functions, $S$ and $T$, actually finding a value of $M$ that has the property we want is a skill that comes only with practice (and perhaps a little graphing). And for some pairs of functions, it’s possible that either $S$ is eventually greater than $T$, or that $T$ is eventually greater than $S$, or neither. Graphing is often a good way to guess which of these you should try to prove.

### 6 Challenge Problem: Left-Max (Optional)

The right-max procedure has a natural parallel left-max, in which we replace each element of a list by the largest element encountered so far, reading left-to-right.

**Example:**

```
(left-max (cons 5 (cons 1 (cons 3 empty)))) =>
(cons 5 (cons 5 (cons 5 empty))
```
See if you can figure out a way, using only ideas we’ve seen so far in class (and only built-in procedures that have been explicitly allowed!) to solve this problem. Again, it’s perfectly OK to use max2, or indeed, any other procedure that you write, to help you get to an answer.

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)