Contents

1 Odds Only 2
2 Increment All 3
3 Function Growth Revisited 3
4 Comparing Functions “up to constants” 5
5 Challenge problem: splitting up sentences 8

Objectives

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

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

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, Function Growth Revisited, and Comparing Functions “up to constants”, 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, as follows (e.g., function-growth.txt corresponds to Function Growth Revisited):

- README.txt
- odds-only.rkt
- increment-all.rkt
• function-growth.txt
• comparing-functions.txt

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 CS-department email address, and optionally with a message to the person grading explaining anything peculiar about the handin. For example:

README.txt:
jhughes@cs.brown.edu
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 zip your hw03 directory into a file hw03.zip.

Introduction

One of the fundamental tools of functional programming is recursion. For example, perhaps you 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 do not want. In this homework, you will 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. For each procedure, you are required to do the following:

1. Data definition
2. Examples of the data
3. Type signature
4. Call structure
5. Specification
6. Code
7. Test cases
8. Run the program

Recursive Diagrams: For this homework in particular, we expect you to write recursive diagrams before you start coding, in order to help you understand the problems. If you get stuck on a problem, draw at least two recursive diagrams and bring these to discuss the problem with a TA.

For more information, you can refer to the pinned Piazza post about the style guide.
Problems

1  Odds Only

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 integers, aloi, and produces (i.e., outputs) another list of integers that contains only the odd numbers in aloi, in their input order. Use the predicate odd? to determine whether the number is odd or not.

Examples:

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

2  Increment All

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

Examples:

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

3  Function Growth Revisited

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 HW 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

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

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 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.

### 4 Comparing Functions “up to constants”

**Warning:** there’s a fair bit of text here, but very little work to do at the end!

We now continue the theme of comparing functions of the form

\[
    f : \mathbb{N} \to \mathbb{R}_+,
\]

i.e., the only functions we’ll discuss take natural numbers to positive reals.
Again, we consider this narrow class of functions because they’re the ones that get used in measuring the performance of computer programs. If you use your spreadsheet to sort a list of 200 numbers, it takes some positive number of seconds. In this case, the size of the problem (200) is the value $n$, and the time taken is a value associated to $n$. We could write

$$S : \mathbb{N} \to \mathbb{R}_+ : n \mapsto \text{The time taken to sort } n \text{ numbers using your spreadsheet program}$$

as a way of defining a function $S$. The problem here is that if you run your spreadsheet program on someone else’s computer — possibly newer and faster than yours — it’ll run in less time when sorting $n$ numbers. The corresponding function, for your friend’s computer — let’s call it $T$, might have the property that

$$T(n) \leq 0.3S(n), \text{ for all } n \in \mathbb{N}. \quad (2)$$

That factor of 0.3 doesn’t have anything to do with the program; it only depends on the computer. So when computer scientists compare “timing functions” like this, they have to ignore constant multiples like the factor of 0.3. (An alternative is to always compare programs on the same computer. But that’s not feasible, as someone might write a new program 15 years after the original, and the original computer might no longer be around for testing!)

So we’ll say that $g$ dominates $f$ up to constants (we’re defining the phrase “dominates ... up to constants”) if there’s a positive number $c$ with the property that for every $n \in \mathbb{N}$, we have

$$f(n) \leq c \cdot g(n).$$

Put in simpler terms: if you graph $f$ and $cg$ for some positive value of $c$, and find that the graph of $cg$ lies above that of $f$ (by which I mean “above or touching the graph of $f$”), then you can say that $g$ dominates $f$ up to constants. If you graph $f$ and $cg$ for every imaginable value of $c$ and the graph of $cg$ never quite manages to lie above the graph of $f$, then you can say that $g$ does not dominate $f$ up to constants.

As an example of “dominating up to constants,” if

$$f(n) = 2n + 1 \quad \text{and} \quad g(n) = 3n + 1 \quad (3)$$

then $g$ dominates $f$ up to constants. Why? Because I can look at $c = 1$, which is positive, and graph $cg$ and $f$ together, and it’s clear that the graph of $cg$ (which is just the graph of $g$) lies above the graph of $f$. (Once again: by “lies above”, I’m including the idea that the graphs of $cg$ and $f$ might touch, but not cross),

How did I know to pick $c = 1$? I drew the two graphs and noticed that $g(n)$ was already above $f(n)$ for every $n \in \mathbb{N}$ except 0, and at 0, it was touching, so it was simple!

What about the case

$$f(n) = 2n + 1 \quad \text{and} \quad g(n) = n + 1 \quad (5)$$

instead?
In this case, choosing $c = 3$ will show that $g$ dominates $f$ up to constants. (Many other values of $c$ would also work — I just thought up a single example.) You can check this claim with Desmos or with some algebra.

The phrase “dominates up to constants” sounds a little like “is greater than”, and you might expect it to be similar to “greater than.” It’s not.

For instance, looking at “greater than” for numbers, the number 4 is not greater than the number 4. By contrast, the function defined by

\[ f(n) = n + 3 \]

dominates up to constants the function $f$. That’s right: $f$ dominates $f$. And the constant $c = 2$ shows it: just graph $n \mapsto n + 3$ and $n \mapsto 2(n + 3)$ with Desmos to see that. (The constant $c = 1$ also shows it!)

In fact, for any function $h : \mathbb{N} \to \mathbb{R}_+$, it turns out that $h$ dominates $h$ up to constants. So the idea that this is similar to “greater than” is definitely misleading.

Let’s look at two functions where one does not dominate the other up to constants:

\[ f(n) = n^2 + 1 \]
\[ g(n) = 3n + 2 \]

No matter what positive constant $c$ you multiply $g$ by, the graph of $f$ will end up above the graph of $cg$ at some point.

Let me say that explicitly: I’m claiming that for any positive $c$ that you choose, there’s a natural number $k$ with the property that

\[ f(k) > c \cdot g(k). \]

You can verify this, for any particular value of $c > 0$, by playing with Desmos, but that’s not a proof that it works for every $c$. It’ll turn out to be a pain to write a proof of this claim, but quite easy to prove something much more general, which we’ll do in class. So for now, we’ll go with the “try it in Desmos” approach to determining whether such a value of $c$ exists.

The actual work for you in this problem involves understanding the definition above clearly enough to compare a few functions. These are

\[ f(n) = 2n + 1 \]
\[ g(n) = n^2 + 1 \]
\[ h(n) = n^2 - n + 1 \]

**Task:** Make a table that looks like this

<table>
<thead>
<tr>
<th></th>
<th>$f$</th>
<th>$g$</th>
<th>$h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f$</td>
<td>$c=1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$g$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$h$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
and fill it in as follows: each table cell is labeled by one function in its row and one in its column. The top right cell, for instance, is labelled by row-function \( f \) and column-function \( h \).

If the row-function dominates the column-function up to constants, you should write in \( c = \ldots \), filling in the smallest value of \( c \) that demonstrates this dominance. As you can see, I’ve done so for the \( f \) vs. \( f \) entry in the upper left.

If the row-function does not dominate the column-function up to constants, you should put an asterisk (*) in that cell.

For the cases where you place an \( c \)-value in a cell, you can find the value of \( c \) by trial and error using Desmos. In trying to show that the function \( e \) dominates the function \( d \), you’re looking for the smallest \( c \) such that the graph of \( ce \) is above-or-touching the graph of \( d \). (In general, we don’t ask for the smallest \( c \) that works; any \( c \) within that works is good enough. But finding the smallest \( c \) demonstrates your understanding of the definitions and makes the problem easier to grade.)

Important note: although Desmos graphs the functions as though they were defined on all of \( \mathbb{R} \), our functions are only defined on \( \mathbb{N} \). So the graph of one need only be above the graph of the other at nonnegative integer values of \( n \).

Note: You’ll get partial credit for correctly placing asterisks and \( c \)-values, and full credit for also correctly choosing the smallest possible \( c \)-values.

5 Challenge problem: splitting up sentences

In the second CS17 project, Eliza, we’ll be working with sentences, like “Hi, how are you today?” We’ll call these “utterances” because sometimes they’ll be incomplete sentences, like “I feel kinda” or multiple sentences, like “I’m not sure. Happy? Dizzy? Unregulated?”

For the main algorithm in the project to function, an utterance like “Hi, how are you today?” needs to be split into something I’ll generically call words, by which I mean “contiguous runs of non-punctuation, non-blank characters or individual pieces of punctuation.” So the words in the sentence I wrote are these:

```
"Hi"
","
"how"
"are"
"you"
"today"
"?"
```

We’ll also need to be able to assemble a list of words back into a sentence, using the ordinary U.S.-English form in which punctuation is not preceded by a blank, but is followed by one, except at the very end of an utterance. That last rule is one I’m using for this assignment, because it happens to be convenient. That means that if the text is something like

```
"I'm" "bored" "." "How" "are" "you" "today" "?"
```

we’d want to convert it to
"I'm bored. How are you today?"

See how there's no blank after the last question-mark, but there is a blank after the mid-utterance period?

In this problem, you're going to try to attack the algorithmic essence of these two problems, but rather than have you mess with strings (which are not Racket's strongest suit), I'm going to abstract the problem, and have you work with lists of ints.

Here's the idea: the integer 0 represents a blank. The integer 1 represents any text-character, like a through z and A through Z and things like apostrophes or hyphens that also appear within words. The integer 9 represents a punctuation mark. And an entire “utterance” will be encoded as a list of these integers. So the first sentence, "Hi, how are you?" would be represented by a list of ints, which I'm going to write immediately beneath the text:

Hi, how are you?
1190111011101119

To be clear, what I mean is that the input to the procedure you'll be writing is a list that's something like

(cons 1 (cons 1 (cons 9 (cons 0 ... empty))...)...)

Your job is to convert this into a sequence of words, each of which is represented by an int list. For this sample, the desired output would be

(cons
 (cons 1 (cons 1 empty)))  <---- Hi
 (cons
  (cons 9 empty))          <---- ,
 (cons
  (cons 1 (cons 1 (cons 1 empty)))) <---- how
 (cons
  (cons 1 (cons 1 (cons 1 empty)))) <---- are
 (cons
   (cons 1 (cons 1 (cons 1 empty)))) <---- you
 (cons
    (cons 9 empty))           <---- ?
                           (empty))))))

where the stuff with the arrows on the far right is to help you see what parts of the original sentence each piece of the output corresponds to. It's not part of the actual Racket expression. (It also makes it tough to cut-and-paste this into Racket; sorry.)

Task: Write a procedure, split, that consumes an int list of the form above, and produces an (int list) list, as described above. Use the design recipe, but you may, in the input/output specification, say this:

output: an (int list) list in which each int list corresponds either to a word or to punctuation, as described in the homework handout
rather than rewriting this whole mess!

**Task:** Write a procedure, assemble, that does the opposite of split: it takes an (int list) list, in which each int list represents either a word or punctuation, so the int zero never appears, and assembles them into a single int list, where each word is followed either by a blank or no-blank-but-then-a-punctuation-mark, and each punctuation-mark is followed by a blank, except that the very last entity (word OR punctuation mark) is followed by no blank at all.

Note that the (int list) list might correspond to just a few words without punctuation, as in an e e cummings poem, and even in this case, the final assembled utterance should have no blank at the end. The input might also consist of multiple sentences, and the ending punctuation of each should be followed by a blank... except for the very last one, of course.

**Task:** What should \(\text{split (assemble word-list)}\) be? What about \(\text{assemble (split utterance)}\)? Are you sure of that last answer?

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]