

Section Notes 5

CS51—Spring 2009

Week of March 2, 2009

Outline

1. Higher order functions review
2. Complexity and Big-O
3. Induction
4. Exam sample question
5. Design note

1 Goals for today:

At the end of today's section, you should be able to do the following:

1. Brag to your friends that your mind is of a higher order... and mean it.
2. Explain to a non-technical person how we analyze the complexity of an operation.
3. Write a recursive function (you already know how to do this).
4. Prove that your recursive function is correct.
5. Extract a recurrence relation for your function.
6. Determine the runtime of your function based on the above.

This week's section will revisit higher order functions and then will focus on what we can learn by analyzing our programs. The former is a tool for writing programs, the latter is a tool for determining that they are correct and efficient.

2 Higher order functions

A *higher-order function* is a function that takes functions as arguments. We will focus on `foldr`, `filter`, and `map`, which implement common patterns in functional programming.

2.1 Fold

Reviewing `foldr`: We usually think of `foldr` as combining the elements of a list to produce a non-list, for example computing the sum of a list of numbers. However, given the right function, `foldr` can also build lists. This effect is achieved by providing a combining operation and a base value that will be used to process the input list. When we call `foldr` we provide three arguments: a binary function we will call `new_cons`, a value we will call `base_case`, and our input list `lst` (i.e. `(fold new_cons base_case lst)`).

The best way of thinking about `foldr` is to envision deconstructing a list and replacing the cons holding the list together with the intended function. That is, the following list `(define lst (list 1 2 3 4))` is actually: `(cons 1 (cons 2 (cons 3 (cons 4 empty()))))`. The base case of `fold` applies where the empty list would be located. For example, summing `lst` would yield 10, which we obtained by constructing the following replacement `(+ 1 (+ 2 (+ 3 (+ 4 0))))`. We can see that 0 is the base case. Therefore, the function `sum` would be written as follows: `(define (sum lst) (foldr + 0 lst))`.

The function that replaces the cons can actually still be cons, which would construct a new list. In addition, the function can be written as a lambda that takes 2 arguments - the first of the list that's returned and the rest of the list that's returned. Of course, neither of these arguments have to be explicitly used in the lambda function; they just have to be written as part of the function because `foldr` expects the lambda function to take 2 arguments. Why can't `foldr` work otherwise?

Here we will use it to write `flatten`. The `flatten` operation takes a list of lists:

```
((1 2) (3 4) (5 6) (7 8))
```

and removes the nesting, leaving us with just the "leaves":

```
(1 2 3 4 5 6 7 8)
```

We can do this by folding with a list-building function. Which one do we need?

Implement `flatten`:

```
(define (flatten lst)
```

2.2 Sorting with filter

The *quicksort* sorting algorithm lends itself nicely to an implementation based on `filter`.

The idea of *quicksort* is to pick a *pivot element* that we guess is the median element of the list. Then we break the problem down by getting a list of elements less than the pivot, and a list of elements greater than the pivot. We sort the two pieces recursively, then get the final result by smooshing together the sorted smaller elements, the pivot, and the sorted larger elements, in that order. If we assume the pieces were sorted properly (using an inductive hypothesis), it is not hard to see that the final list is sorted as well.

How do we guess a good pivot? There is some debate regarding the best way to do this, but for now we will arbitrarily pick the first element. We will also use `let` to give the pivot a name within the procedure. For simplicity, start by assuming we'll sort lists of numbers with no duplicates.

```
(define (qsort lst)
  (if (null? lst) '()
      (let ((piv (car lst)))
        ; your code here!
```

2.2.1 Sort enhancements

Consider the following:

- What if we want to sort lists of arbitrary values? Modify `qsort` so it is parameterized by the comparison function to use.
- What if we want to sort values by comparing something other than the values themselves? For example, how could we sort a list of lists of numbers by their first elements?
- What happens if we give our `qsort` a list with duplicated elements? Fix the function to work in such a case.

2.3 More examples

The `flatten` function we defined earlier is only able to flatten one level of nesting. Write a version that can flatten *all* the structure of a list, no matter how deeply nested:

```
(define (deep-flatten lst)
```

Use `fold` to write a function that converts a list of digits to a number. For example, `(list 1 2 3 4)` would be 1234.

```
(define (make-num lst)
```

Convert a list of Fahrenheit temperatures to a list of Celsius temperatures using the `map` function.

```
(define (temp-convert lst)
```

Divide a list into nested sublists. That is, given this input `(list 4 6 2 3 7 1 5 1 2 3 8)`, return `(list (list 4) (list 6) (list 2) (list 3) (list 7) (list 1) (list 5) (list 1) (list 2) (list 3) (list 8))`

```
(define (bucket lst)
```

3 Complexity and Big-O

The runtime of algorithms is often complex and hard to express. Big-O notation provides a way for us to simplify the way we think about the complexity of algorithms and allows us to easily compare across different algorithms. If two algorithms have the same Big-O bound, we can group them into time-complexity classes, which are broad categorizations of the time it takes an algorithm to run.

For example, take this simple algorithm for sorting a list.

1. Count the number of items in the unsorted list. (n)
2. Create a new list to hold sorted items. (c)
3. Starting from the first item:
 - a. Take the item at the beginning of the unsorted list. (n)
 - b. Find the place where it should go in the sorted list. (n)

Thus, the runtime of this algorithm is $f(n) = n^2 + n + c$. What is Big-O of $f(n)$?

Big-O is an estimate of the asymptomatic bound of a function. That is, what is approximately the running time of the algorithm on very large inputs? Because we are concerned with the running time on large inputs, we can disregard terms of lower orders; these terms have lesser influence when the size of inputs increase. In addition, we can disregard the coefficients in front of terms, since constant multiples depend on the speed of the processor and do not affect the underlying efficiency of the algorithm. For example, $f(n)$ can be reduced to n^2 ; therefore, its Big-O relationship is $f(n) = O(n^2)$.

The broad time-complexity classes we're interested in are: logarithmic time ($O(\log n)$), constant time $O(c)$, polynomial time $O(n^x)$, and exponential time $O(x^n)$. These are ordered from most time-efficient to least time-efficient. Why is this so? Imagine graphing equations of these types, and you can see which ones grow at a higher rate and eventually surpass the other equations.

1. Which of the following is the most efficient? Can you think of any examples of algorithms that run in these times?
 - (2^n)
 - $(\log(n))$
 - (n^3)
 - $(4n^3)$
2. In how many steps would the binary search algorithm halt if it were to search for the value 17, in the set $S = 2, 3, 5, 7, 11, 13, 17, 19, 23$?
3. What is the big O of the algorithm mentioned in the previous question?
 - $(\log(n))$
 - (n)
 - (c)
 - $n \log(n)$

4 Induction

Induction is a mathematical method that we use as Computer Scientists to argue the validity of a statement or conjecture. Before CS51, you may have encountered inductive proofs in math classes, perhaps to prove statements about sets of numbers (in particular, the natural numbers). In this class, we use induction as a means of proving properties about programs.

4.1 The Basics

As proficient Schemers, induction may seem eerily familiar to you. Inductive proofs have the following basic structure:

1. Determine the property to be proven.
2. Establish that the property holds for a base case.
3. Propose an inductive hypothesis that we will assume to be true.
4. Prove the inductive step (i.e. show that the inductive hypothesis holds for the “next” item or element).

Why does this seem familiar? Well, the programs that we have been writing this semester have required you to formulate base cases and recursive cases. Your recursive case is written making an assumption about the correctness of calling the function recursively. This leap of faith is similar to the mental leap that needs to be made when writing inductive proofs.

Turning the abstract into the practical, keep in mind Ramin’s recipe for writing inductive proofs throughout the rest of this section:

1. Write down the variable you’re doing induction on.
2. Write down the property to be proven (e.g. the function $P[x]$).
3. Prove your base case (e.g. prove $P[0]$)
4. Write down your inductive hypothesis that you assume to be true (e.g. $P[k] = \text{true}$), and prove the next step (e.g. $P[k+1] = \text{true}$).

4.2 Proving Properties of Functions

Before jumping into a proof, let’s warm up by writing a relatively simple function in Scheme:

```
; recursive sum-1-to function
(define (sum-1-to n)
```

```
)
```

Think about your two cases. At this point in the semester you probably have taken a simple recursive function like this for granted. However, before we move on, consider that one of the cases depends on `sum-1-to` returning the right answer when called recursively. You might consider this a recursive hypothesis. Now, let’s simplify this function by rewriting it so we don’t need to make that recursive call at all:

```
; non-recursive sum-1-to function
(define (sum-1-to n)
```

```
)
```

It looks like there are multiple approaches to solving this problem. This raises the question of how we might verify that our approaches are correct and equivalent. Let us see how we can prove that our first function is correct? More precisely, how would we prove that the recursive function `sum-1-to(n)` returns the sum of the integers between 1 and `n` (inclusive), for all $n \geq 1$? Let's use the inductive proof technique outlined above.

Let's start by identifying the variable we are inducting over and the property that we are proving:

variable =

P[] =

Now, let's pick our base case and demonstrate that our property holds for it:

Lastly, we need to write down our inductive hypothesis and prove our inductive step:

In proving your inductive step, you needed to rely on your inductive hypothesis. Much like our so-called recursive hypothesis above, our proof relies upon a property of the function when passed some value `k` as an argument. Moreover, just like a recursive function, the soundness of our proof depends on the presence of a base case. Without this we logically break down a call with `k+1` as an argument into an expression that has a call with `k`, and then into `k-1`, etc, but we never reach "bottom". Lucky for us our brains don't go into infinite loops when faced with an inductive proof that is missing its base case!

4.3 Recurrence Relations

Above we have seen how we can use inductive proofs to establish that a property holds for some set of inputs to a function. Theoretical analysis of a recursive function can also yield us insight into the amount of effort that will be required to compute a function for some input, specifically the O or Big-O of the function.

To extract a recurrence relation from a function that describes its runtime, we often break it down according to its cases. Each case can typically be represented with its own equation, where we will map constant-time operations to constant values and function calls (recursive or otherwise) to applying mathematical functions. For example, consider the recurrence relations that we extract from our recursive `sum-1-to` function above:

$$T_{sum-1-to}(0) = c$$

$$T_{\text{sum-1-to}}(n) = c + T_{\text{sum-1-to}}(n - 1)$$

If we expand out the latter function we ultimately determine that this function is in $O(n)$. What we discover is that we have a mathematical analysis of a function in Scheme that we can use to make statements about its runtime.

4.4 Useful Recurrence Relations

In lecture, we saw the following basic recurrence patterns that you will likely find helpful as you analyze functions:

$$\begin{aligned} T(n) &= c + T(n - 1) \in O(n) \\ T(n) &= k * n + T(n - 1) \in O(n^2) \\ T(n) &= c + T(n/2) \in O(\log n) \\ T(n) &= k * n + T(n/2) \in O(n) \\ T(n) &= k * n + 2T(n/2) \in O(n \log n) \\ T(n) &= c \in O(1)^1 \end{aligned}$$

Make sure that you have at least an intuitive sense of why these statements are true. You will need to be comfortable using these facts in analyzing both programs that you write and those that you might be presented as part of an exam or problem set question.

Construct a recurrence relation for our non-recursive version of `sum-1-to` and solve for $T_{\text{sum-1-to}}(n)$.

5 Sample Exam Question

The material that we have covered thus far is likely to show up in an exam. Here's an example of what this type of problem would look like (bear in mind, though, that the actual exam question is likely to be more complex):

5.1 Scheme

Write a fibonacci function that takes in a natural number, `n`, as an argument and returns the n 'th number in the fibonacci sequence. Recall that the fibonacci sequence is recursively defined such that each successive value is equal to the sum of its two immediate predecessors. That is,

$$\begin{aligned} \text{fib}(0) &= 1 \\ \text{fib}(1) &= 1 \end{aligned}$$

¹Though this didn't appear in lecture, this is also an important pattern!

$$fib(n) = fib(n - 1) + fib(n - 2)$$

Thus, the first 8 numbers in the fibonacci sequence are: 1, 1, 2, 3, 5, 8, 13, 21. Complete the function below:

```
(define (fibonacci n)
```

5.2 Proof

Prove inductively that your fibonacci function is correct. Make sure that you follow the recipe discussed in class and make sure that the grader can clearly identify each part of your proof.

5.3 Recurrence Relation

Write appropriate recurrence relations to represent the running time of your fibonacci function. Make sure that all cases are covered.

5.4 Big-O Notation

Solve the recurrence relations that you found in the previous part to determine an upper-bound on the running time of this function.

6 Design Note

Having trouble coming up with a name for a function? This is a common symptom² that may suggest the presence of a larger design issue. In general, think hard about the name of a function, as well as its appropriate inputs, and output before sitting down to write the body. A good name should describe in human-readable terms the intent of this function. Also proper naming will often reveal other design issues, such as breaking the abstraction barrier or lack of generality in your code.

Here are some examples of bad function names, can you identify a problem with each?

```
(define (generate-3-tuples-from-list lst) ...
(define (clique-helper-2 c l a) ...
(define (mystery n x) ...
(define (n-tree-car t) ...
```

7 Answers

Section 2

- ```
(define (flatten lst)
 (foldr append empty lst))
```
- ```
(define (qsort lst)
  (if (null? lst) '()
      (let ((piv (car lst)))
        (append (qsort (filter (lambda (x) (< x piv)) lst))
                (list piv) (qsort (filter (lambda (x) (> x piv)) lst))))))
```
- Sorting duplicate items:

```
(define (qsort lst)
  (if (null? lst) '()
      (let ((piv (car lst)))
        (append (qsort (filter (lambda (x) (> x piv)) (cdr lst)))
                (list piv) (qsort (filter (lambda (x) (<= x piv)) (cdr lst))))))
```
- ```
(define (deep-flatten lst)
 (foldr (lambda (x rest) (if (number? x) (cons x rest) (append (deep-flatten x) rest))) '() lst))
```

---

<sup>2</sup>These symptoms are often referred to as “code smells”. If you Google for the term, you will discover that there is a large community that focuses on how to identify and address problems in code.

- (define (make-num lst)
 (foldl (lambda (digit accum) (+ (\* 10 accum) digit)) 0 lst))
- (define (temp-convert lst)
 (map (lambda (temp) (\* (/ 5 9) (- temp 32))) lst))
- (define (bucket lst)
 (map list lst))

## Section 4.2

```
; recursive sum-1-to function
(define (sum-1-to n)
 (if (= n 1)
 1
 (+ n (sum-1-to (- n 1)))))
```

```
; non-recursive sum-1-to function
(define (sum-1-to n)
 (* (/ n 2) (+ n 1)))
```

Recipe for proving that recursive `sum-1-to` is correct:

1. Variable is  $n$ , the input, whose type is a positive integer.
2.  $P[n]$  = the value of `(sum-1-to n)` equals the summation of the integers between 1 and  $n$ , inclusive.
3. Base case: show that  $P[1]$  holds (i.e. returns 1), by the evaluation rules, `(sum-1-to 1)` is the body with the variables  $n$  replaced by 1:

```
(if (= 1 1)
 1
 (+ 1 (sum-1-to (- 1 1))))
```

The predicate evaluates to true, and so the value of `(sum-1-to 1)` is 1.

4. Inductive hypothesis: assume  $P[n]$  holds for arbitrary  $n$ . Now prove  $P[n + 1]$  from  $P[n]$ . By the evaluation rules, `(sum-1-to n+1)` is:

```
(if (= n+1 1)
 1
 (+ n+1 (sum-1-to (- n+1 1))))
```

The predicate evaluates to false, so we must now show that: `(+ n+1 (sum-1-to (- n+1 1)))` evaluates to the sum of the number from 1 to  $n+1$ .

By the evaluation rules, this expression is: `(+ n+1 (sum-1-to n))`.

By the induction hypothesis,  $P[n]$  holds, and therefore, this expression is `(+ n+1 (sum of integers 1 to n inclusive))`.

By simple arithmetic, we see that adding  $n+1$  to the sum of integers from 1 to  $n$  yields the sum of integers from 1 to  $n+1$ .

## Section 4.4

A recurrence relation for our non-recursive version of `sum-1-to` would look like:

$$T_{\text{sum-1-to}}(n) = T_*(T_-(na), T_+(nb))$$

where  $a$  and  $b$  are constants. However, if we assume that arithmetic operations are in  $O(1)$ , this means that  $T_*$  is not a function of its arguments, and therefore:

$$T_{\text{sum-1-to}}(n) = k$$

for some constant  $k$ . Thus, we would say that our non-recursive version of this function is:

$$T_{\text{sum-1-to}}(n) \in O(1)$$

## Section 5

To check your answers to the sample exam problem, please contact a TF (either in person at office hours or over email).