

Section Notes 10

CS51—Spring 2009

Week of April 19, 2009

1 Outline

1. Dynamic Programming
2. Segmented Least Squares
3. Tail Recursion

At the end of this section, you should know how Dynamic Programming can be used to reduce the complexity of problems like Segmented Least Squares, be able to eliminate unnecessary stack usage in some recursive functions.

2 Dynamic Programming

Despite the fact that you are in a Computer Science course, the concept of *Dynamic Programming* is not related, at a fundamental level, to computer programming or programming languages. Rather, Dynamic Programming is a form of optimization that just so happens to be well suited to being implemented on a computer. Specifically, this optimization provides an efficient method for solving a certain class of problems: those that contain optimal substructure and overlapping subproblems. In this section we will discuss the theory behind Dynamic Programming as well as review an example of one application.

2.1 What problems can it solve?

Dynamic Programming can be used to solve a particular class of problems. Specifically, it is used to optimize problems that are solved in terms of smaller “subproblems”. These problems share two properties:

1. Optimal Substructure - finding an optimal solution to subproblems can be used to find an optimal solution to whole problem. Or, viewed from the other direction, the optimal solution to the whole problem contains optimal solutions to any subproblems. For example, if we use DP to guide us between cities that are represented as a graph, then if we know that the shortest path from Boston to Miami goes through New York, then we know that the path generated between New York and Miami as part of the larger solution is also the shortest path between these two nodes in our graph.
2. Overlapping Subproblems - a form of divide and conquer, the same smaller problem is solved for different parameters. However, unlike with basic recursion, these smaller problems overlap. For example, $\text{Fib}(n)$ is defined in terms of $\text{Fib}(n-2)$ and $\text{Fib}(n-1)$. The definition of $\text{Fib}(n)$ overlaps with the definitions of $\text{Fib}(n-1)$, which is defined in terms of $\text{Fib}(n-2)$ and $\text{Fib}(n-3)$.

2.2 Memoization

One key idea in Dynamic Programming is that problems that consist of Overlapping Subproblems can often be solved efficiently by *memoizing* the answers to subproblems. As we discussed last week in section, memoizing is a technique that we can use to reduce the running time of a program. Specifically, memoization works by saving the result of function calls into a table to avoid re-computing the result each time the same function is called with the same arguments.

A classic example of a function that could benefit from memoization is `fib`, the function that computes the n 'th Fibonacci number. To see this in action, let's consider the simple, un-memoized `fib` function:

```
(define (fib n)
  (cond [(= n 0) 1]
        [(= n 1) 1]
        [else (+ (fib (- n 1)) (fib (- n 2)))]))
```

The problem with this implementation can be demonstrated by analyzing what function calls will be made in some examples:

```
(fib 5) → (+ (fib 4) (fib 3))
(fib 4) → (+ (fib 3) (fib 2))
(fib 3) → (+ (fib 2) (fib 1))
(fib 2) → (+ (fib 1) (fib 0))
(fib 1) → 1
(fib 0) → 1
```

Note that computing `(fib 5)` depends on computing `(fib 3)` twice: once directly and once as part of computing `(fib 4)`.

To avoid recomputing, we can construct a table that maps from inputs to `fib` to the corresponding answer. Although memoization can be solved generally using higher order functions in Scheme that we will see shortly, we will start with memoizing our `fib` function by hand. Let's consider how you would use a Hash Table¹ in Scheme to implement a solution that works for `fib`

```
(define fibht (make-hash))

; INSERT COMMENT HERE
;
;
;
;
(define (hash-ref! ht k c)
  (hash-ref ht k (lambda () (let ([v (if (procedure? c) (c) c)])
                              (begin (hash-set! ht k v) v))))))

(define (fib n)
```

As promised, there is also a higher-order function called `memoize` that can be used to memoize calls to almost any Scheme function. `memoize` takes a function as an argument and returns a function that exhibits the same behavior as the argument except that the inputs and return values are memoized in a table for later use. You will use this function in your implementation for Project 2c.

¹See <http://docs.plt-scheme.org/guide/hash-tables.html> for more information

```

(define (memoize function)
  (let ([table (make-hash)])
    ; Funny lambda syntax: takes args as a list
    (lambda args
      (dict-ref table
        args
        ;; If the entry isn't there, call the function.
        (lambda ()
          (let (; Call the function
                [result (apply function args)])
            (dict-set! table args result)
            result))))))

```

3 Dynamic Programming and Fitting

We have seen many examples of how we might fit data to a line in lecture. We will look at *Segmented Least Squares* in more depth here. Like Least Squares, Segmented Least Squares can provide us with a method of evaluating the error associated with a fit. The difference between Least Squares and the Segmented version is that the latter is used when we are considering using multiple lines to fit to our datapoints.

In lecture and in the included figure, we saw examples of data plots that appeared to fit to one line for some of the points, but then veered off, suggesting that an appropriate fit might require two lines or more lines.

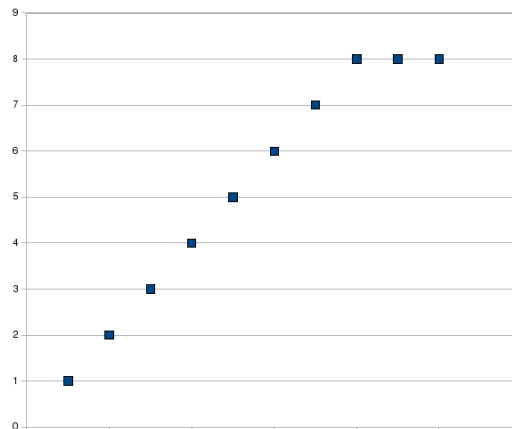


Figure 1: Data points that appear to fit multiple lines.

Recall that finding a good fit with Least Squares means minimizing an error function. Specifically, the Least Squares error function was defined as:

$$Err(m, b) = \sum_i (y_i - (mx_i + b))^2$$

When fitting multiple line segments, we want to minimize:

- The total error (i.e. the sum of the sum of squared errors in each segment).
- The total number of line segments.

Given these two things that we want to minimize, we fix a constant factor that represents the relative importance of minimizing the total number of line segments. So, the cost function that we try to minimize

in Segmented Least Squares is the total error value of our segments plus our constant, c , times the number of line segments.

Using the cost function described above we can break down this problem into subproblems that we can then optimize. To do this, let's start by refreshing some of the notation that Ramin used in lecture:

- $OPT(j)$ = minimum cost (accounting for sum of squares error and lines) for fitting lines to the first j points.
- $e(i, j)$ = minimum sum of squares error determined based on the points i through j .

Based on this notation, we know that $OPT(j)$ is 0 if $j = 1$. But, for any other value of j , it is:

$$OPT(j) = \min_{i < j} (e(i, j) + c + OPT(i - 1))$$

where i is the value between 1 and j that minimizes the function. In other words, computing the minimum cost for points 1 through j consists of minimizing the Least Squares error based on i through j , adding the constant penalty for an additional line, and adding the value of OPT through the last break (i.e. the right-most line change), which happens at $i - 1$.

What this means is that we start by doing a search over possibilities for the final line segment (the right-most segment). This search runs by starting at some point, i , and going through j . For a given i , we rely on the recursive call to compute the optimal fit for the data points from 1 to $i - 1$ and then combine that optimal's fit error with the error of the line that we fit from i to j .

This optimization maps in a relatively straight-forward way to a Scheme implementation. As we would expect, it benefits from memoization. To see an example implementation, check out the Segmented Least Squares scheme file on the course website.

4 Tail Recursion

We have seen the power of recursive functions to express looping constructs. In Scheme, recursive function calls take the place of constructs like `for` or `while` in languages like C. While recursive functions have a simple elegance stemming from their mathematical origin, we have also encountered practical limitations. Most notable, recursive functions make use of the “stack” to store per-iteration information. In practice, this limits the number of iterations that we can perform before we run out of stack space. Consider the following function:

```
; sum-to-n adds up the natural number from 0 to n, inclusive, and returns the
; result.
(define (sum-to-n n)
  (if (= n 0)
      0
      (+ n (sum-to-n (- n 1)))))
```

This function is in $O(n)$ both in terms of its runtime growth and its space growth. Our function's use of space increases with n because a larger n requires stacking up more recursive calls. For example, a call to `(sum-to-n 5)` can't return until `(sum-to-n 4)` returns, which will not return until `(sum-to-n 3)` returns, etc. Our evaluation ends up looking something like:

```
(sum-to-n 5) → (+ 5 (sum-to-n 4))
(sum-to-n 4) → (+ 4 (sum-to-n 3))
(sum-to-n 3) → (+ 3 (sum-to-n 2))
(sum-to-n 2) → (+ 2 (sum-to-n 1))
(sum-to-n 1) → (+ 1 (sum-to-n 0))
(sum-to-n 0) → 0
```

This presents a problem because we have a limited amount of memory and in turn a limited amount of stack space. The memory limit for DrScheme is reported just under the Welcome message (and is probably 128 megabytes in your configuration). Like many limits that we encounter in programming, 128,000,000 bytes seems like a lot, until we hit it. For our function above, you will hit this limit before n reaches 10 million. Feel free to try it for yourself to see what happens when “run out of memory”.

By comparison, what happens with the following snippet of C code?

```
int max = 10 * 1000 * 1000; /* 10 million */
int sum = 0;

int i;
for (i = 0; i < max; i++) {
    sum += i;
}
```

While it may seem that we have encountered a flaw in Scheme’s approach to looping, there is an answer. It turns out that there is a class of recursive function calls that the Scheme interpreter can rewrite to resemble the C for loop above. In particular, these are what we call *tail calls*. A tail call is a function call that is the last operation that is required of a function. Note that in `sum-to-n`, the recursive call was not the last operation. In that function, after the recursive call, we add n to its result. The claim is that if we rewrite our function so that our recursive call is the last operation, then Scheme can cleverly compile our function into something more space-efficient.

Let’s rewrite our function with a helper and an accumulator:

```
(define (sum-to-n2 n)
  (letrec ([helper
            (lambda (n ac)
              (if (= n 0)
                  ac
                  (helper (- n 1) (+ n ac))))])
    (helper n 0)))
```

We can try the same experiments as above and find that this function doesn’t have the same problem. Why can this code be optimized into a more efficient form? The key insight is that the stack frames from the recursive calls of this recursive function can be discarded as soon as the call enters into the next recursive call. Consider what happens with a call to `(sum-to-n2 5)`:

```
(sum-to-n2 5) → (helper 5 0)
(helper 5 0) → (helper 4 5)
(helper 4 5) → (helper 3 9)
(helper 3 9) → (helper 2 12)
(helper 2 12) → (helper 1 14)
(helper 1 14) → (helper 0 15)
(helper 0 15) → 15
```

Notice that when the last call, `(helper 0 15)` is complete, there is no work left to be done. This is in contrast with our earlier example where `(sum-to-n 0)` returned 0, which still required the pending add operations to calculate the final answer. With a tail call, the compiler is free to rewrite the code so that entering the new function does not result in a new stack frame. In the case above, this means that our final call to `helper` will return its result, 15, all the way back to the code that called `sum-to-n2`, without going through any intervening stack frames.

Writing code in a tail-recursive style is an important skill for Scheme developers. Languages that depend on recursive function calls as the core looping primitive require tail call optimization to allow efficient implementation of looping functions. In fact, you will likely find more examples of using an accumulating parameter in previous lecture and section notes. This programming idiom is typically used to ensure that a recursive function is tail recursive.

4.1 Sample Problems

Tail recursion is used to write efficient functions for solving problems in different domains. Try writing the following functions in a tail recursive fashion as practice.

```
; fact computes the factorial function (aka n! or n * (n - 1) * (n - 2) ... * 1)
(define (fact n)
```

```
; fold_left computes a left fold (equivalent to foldl), which recursively
; combines the results of combining all but the last element with the last one.
;
; e.g. (fold_left cons '() '(1 2 3 4)) => (fold_left cons '(1) '(2 3 4)) =>
; (fold_left cons '(1) '(2 3 4)) => (fold_left cons '(2 1) (3 4)) =>
; (fold_left cons '(3 2 1) '(4)) => (fold_left cons '(4 3 2 1) '()) =>
; '(4 3 2 1)
(define (fold_left f base lst)
```

```
; fold_right computes a right fold and is not naturally tail recursive. This
; is because (fold_right f base lst) naturally expands to:
; (f (car lst) (f (cadr lst) (f (caddr lst)... (f (last_element lst) base))))
; That said, this function can also be written to be tail recursive using
; a bit of trickery.
(define (fold_right f base lst)
```

```
; rev takes a list and returns a list with the same elements in reversed
; order. This function is equivalent to the Scheme function reverse.
(define (rev lst)
```

5 Answers

- ```
(define fibht (make-hash))

; hash-ref! looks up k in ht. If it's not found, then this function computes a
; value (thunking c if appropriate) and adds a mapping from k to this value in
; ht.
(define (hash-ref! ht k c)
 (hash-ref ht k (lambda () (let ([v (if (procedure? c) (c) c)])
 (begin (hash-set! ht k v) v))))))

(define (fib n)
 (cond [(= n 0) (hash-ref! fibht 0 (lambda () 1))]
 [(= n 1) (hash-ref! fibht 0 (lambda () 1))]
 [else (hash-ref! fibht n (lambda () (+ (fib (- n 1)) (fib (- n 2)))))]
))
```
- ```
(define (fact n)
  (letrec ([helper
            (lambda (n ac)
              (if (= n 1)
                  ac
                  (helper (- n 1) (* n ac))))])
    (helper n 1)))
```
- ```
(define (fold_left f base lst)
 (if (null? lst)
 base
 (fold_left f (f (car lst) base) (cdr lst))))
```
- ```
(define (fold_right f base lst)
  (fold_left f base (reverse lst)))
```
- ```
(define (rev lst)
 (letrec ([helper
 (lambda (lst ac)
 (if (null? lst)
 ac
 (helper (cdr lst) (cons (car lst) ac))))])
 (helper lst '())))
```