MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001 Structure and Interpretation of Computer Programs
Spring, 2007

Recitation 4, Friday February 16

Order of Growth Problems Dr. Kimberle Koile

1. What are the orders of growth for the find-e procedure?

```
(define (find-e n)
  (if ((= n 0)
      1
      (+ (/ 1 (fact n)) (find-e (- n 1)))))
```

time	$\Theta(n^2)$
space	$\Theta(n)$

calls proportional to n = $\Theta(n)$ for find-e

(fact n): $\Theta(n)$ (find-e (- n 1)): $\Theta(n)$

1 deferred op

$\Theta(n) * \Theta(n) = \Theta(n^2)$ time

the / op doesn't add time or space because it is part of (/ (fact n)) expression + is only one operation.

2. Louis Reasoner is having great difficulty with a procedure he wrote that uses his version of fast-expt. No matter what argument n he gives it, it tells him that n multiplications are required to raise something to the nth power using fast-expt.. He feels fairly certain that's not right. Louis calls his friend Eva Lu Ator over to help. When they examining Louis' code, they find that he has rewritten fast-expt to use an explicit multiplication, rather than calling square.

```
(define (fast-expt b n))
  (cond ((= n 0) 1)
        ((even? n) (* (fast-expt b (/ n 2)) (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

two evaluations now required, each halves the problem (but the two combine to produce n ops)

"I don't see the difference the multiplication could make," says Louis. "I do," says Eva. "By writing the procedure like that , you have transformed the Θ(log n) process into an Θ(n) process." Explain.

.

3. What are the orders of growth for each of these procedures? (Assume n is positive.)
Assume that you have a procedure `divisible?` that returns #t if n is divisible by x.
It runs in $\Theta(n)$ time and $\Theta(1)$ space.

Note that in Scheme, as shown here, procedures can be defined within other procedures.

a.
```
(define (prime? n)
  (define (helper curr n)
    (cond ((>= curr n) #t)
          ((divisible? n curr) #f)
          (else (helper (+ 1 curr) n))))
  (helper 2 n))
```

n steps

n calls

time $\Theta(n^2)$

space $\Theta(1)$

no deferred ops

time: helper is called n times; divisible is called each time $\therefore$ n*n

b. This version is more clever in that the helper procedure runs fewer times.

```
(define (prime-fast? n)
  (define (helper curr)
    (cond ((> curr (sqrt n)) #t)
          ((divisible? n curr) #f)
          (else (helper (+ 1 curr)))))
  (helper 2))
```

n steps

time $\Theta(n^{3/2})$

space $\Theta(1)$

helper will only be called $\sqrt{n}$ times; divisible is $\Theta(n)$ and is called each time $\therefore$ $n^{1/2} * n$

Note that if `sqrt` is slower than `square`, we could write the first clause in the cond statement as (> (square curr) n) .

4. Write a procedure that computes the number of decimal digits in its input and that is linear in space and time in the number of digits: (num-digits 102) => 3 Do not use logs; use the procedure quotient: (quotient 21 5) => 4

```
(define (num-digits n)
  (if (= n 0)
      0
      (+ 1 (num-digits (quotient n 10)))))
```

left for next call

1 is added for digit in ones place; e.g. (quotient 11 10) = 1
+1

5. The procedure s for exponentiation have been written in terms of repeated multiplication. In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication procedure (in which it is assumed that our language can only add, not multiply) is analogous to the expt procedure:

```
(define (* a b)
  (if (= b 0)
      0
      (+ a (* a (- b 1)))))
```

This algorithm takes a number of steps that is linear in b. Now suppose we include, together with addition, operations double, which doubles an integer, and halve, which divides an (even) integer by 2. Using these procedures, design a multiplication procedure analogous to fast-expt that uses a logarithmic number of steps. See explanation in section 1.24 of text.