Topic 4
Orders of Growth

September 2008

Orders of growth

- When a procedure is called, how do time and memory grow as a function of the size of the input?
- Size of an integer = its value
- Size of a string = its length
- Size of a graph structure = # of nodes or # of links

A definition

- Let R(n) = amount of resource (time, memory) used when n = size of input
- R(n) has **order of growth $\Theta(f(n))$ if there exist constants k_1 and k_2 s.t.**
  $k_1 * f(n) \le R(n) \le k_2 * f(n)$
  for all sufficiently large n.

Theta(polynomial)

If R(n) is a polynomial such as
$a_0 + a_1 * n^1 + a_2 * n^2 + a_3 * n^3 + \ldots + a_m * n^m$
then R(n) has order of growth $\Theta(n^m)$.

Recursive factorial

- For both time and memory, recursive fac(n) has order of growth $\Theta(n)$ because the number of steps grows proportionally to the input n.
- R(n) = R(n-1) + k

```
; takes a positive integer and returns its factorial
; (fac 1) = 1; if n>1, then (fac n) = (* n (fac (- n 1)))
(define (fac n)
 (if (= n 1)
    1
    (* n (fac (- n 1)))))
```


Iterative factorial

- For time, ifac(n) has order of growth $\Theta(n)$
- For memory, ifac(n) has order of growth $\Theta(1)$
- $k_1 * 1 \le$ amount of memory $\le k_2 * 1$

```
; iterative version of factorial
; takes a positive integer and returns its factorial
(define (faci n)(ifac 1 1 n))

; helping fn for iterative version of factorial
 (define (ifac val cur-cnt max)
   (if (> cur-cnt max)
      val
      (ifac (* cur-cnt val)
            (+ cur-cnt 1)
            max)))
```

Orders of Growth

- Orders of growth provide only a crude description of the behavior of a process.
- This is still often very useful – especially as numbers (n's) are very large.
- The difference between a process that is linear ($O(n)$) versus $O(n^2)$ can mean the difference between being able to run the algorithm on a particular input and not being able to run it.

Exponentiation

$b^n = 1$ if $n = 0$
$= b * b^{n-1}$ if $n > 0$

```
; raises b to the nth power
; where n is a positive integer
(define (expt b n)
  (if (= n 0)
     1
     (* b (expt b (- n 1)))))
```

$\Theta(n)$ in both time and space.

Can we do better?

- We could do better by developing a procedure that generated a linear iterative process rather than a recursive one.

Iterative exponentiation

```
; computes b to the n
(define (exponent-i b n)
  (ipwr 1 b n))

; val contains intermediate value
; val = b^(n-ctr)
(define (ipwr val b ctr)
  (if (= ctr 0)
      val
      (ipwr (* b val) b (- ctr 1))))
```


Exponentiation

$b^n = 1$ if $n = 0$
$= b * b^{n-1}$ if $n > 0$

For both time and memory, $\Theta(n)$ if process is recursive.

For memory, iterative process can be $\Theta(1)$

Notice: Computing b^8
b * (b * (b * (b * (b * (b * (b * b))))))
But
b^2 = b * b
b^4 = b^2 * b^2
b^8 = b^4 * b^4 I can do the computation in far fewer steps!

This works for exponents that are powers of 2. In general

$b^n = (b^{n/2})^2$ if n is even (NOTE: book has error!)
$= b * b^{n-1}$ if n is odd

Faster code

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


Analysis (watch it run)

- At least every other recursive call has an even input
- $R(n) \approx R(n/2) + k$
- For time and memory, has order of growth $\Theta(\log_2 n)$