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.

$$k_1 * f(n) \leq R(n) \leq k_2 * f(n)$$

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 growh $\Theta(n^m)$.

Recursive factorial

- For both time and memory, recursive fac(n) has order of growth $\Theta(n)$
- R(n) = R(n-1) + k

Iterative factorial

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

Tree recursive fibonacci

- For time, fib(n) has order of growth $\Theta(\varphi^n)$
- For memory, fib(n) has order of growth $\Theta(n)$
- For time: fib(n) = fib(n-1) + fib(n-2)
- R(n) = R(n-1) + R(n-2) + k

$$\frac{R(n)}{R(n-1)} = 1 + \frac{R(n-2)}{R(n-1)} + \frac{k}{R(n-1)}$$

(continued)

$$\frac{R(n)}{R(n-1)} \rightarrow r \qquad r = 1 + \frac{1}{r}$$

$r = \varphi$ $\qquad R(n) \approx \varphi * R(n-1) \qquad R(n) \approx \varphi^n$

(continued 2)

- For memory: R(n) = R(n-1) + k
- Only a fixed amount of memory is needed to store the value of R(n-2), which could be computed before R(n-1).
- R(n) has order of growth $\Theta(n)$

Some general rules

- If $R(n) \approx R(n-j) + k$, $\Theta(n)$
- If $R(n) \approx R(n-j) + k * n$, $\Theta(n^2)$
- If $R(n) \approx R(n-j) + k * n^i$, $\Theta(n^{i+1})$
- If $R(n) \approx r * R(n-j)$, $\Theta(r^{n/j})$
- If $R(n) \approx R(n/j) + k$, $\Theta(\log_j n)$

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 $\Theta(1)$

Iterative exponentiation

```
(define (exponent b n)
  (ipwr 1 b n))
(define (ipwr val b n)
  (if (= n 0)
      val
      (ipwr (* b val) b (- n 1))))
```

A more efficient way

$b^n = (b^{n/2})^2$ if n is even

$= b * b^{n-1}$ if n is odd

(book has error)

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

- 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 $\Theta(\log_2 n)$