Custo de Computação

- tempo...
- espaço...

Concreto $\times$ Abstrato ?

Exemplo 1

```
(define (how-many a-list)
  (cond
    [(empty? a-list) 0]
    [else (+ (how-many (rest a-list)) 1)]))

  (how-many (list 'a 'b 'c))
= (+ (how-many (list 'b 'c)) 1)
= (+ (+ (how-many (list 'c)) 1) 1)
= (+ (+ (+ (how-many empty) 1) 1) 1)
= 3
```

$\Longrightarrow$ Quanto mais longa a entrada mais passos de recursão são necessários.

$\Longrightarrow$ O número de passos entre as recursões é sempre o mesmo.

Função de custo (abstrato)

$\Longrightarrow$ Escolher unidade de medida : chamadas recursivas;

$\Longrightarrow$ Usar tamanho da entrada como variável.

Exemplo 2

```
(define (contains-doll? a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) false]
    [else (cond
    [(symbol=? (first a-list-of-symbols) 'doll) true]
    [else (contains-doll? (rest a-list-of-symbols))])]))

(contains-doll? (list 'doll 'robot 'ball 'game-boy 'pokemon))
(contains-doll? (list 'robot 'ball 'game-boy 'pokemon 'doll))
```

$\Longrightarrow$ O número de passos necessários varia também conforme a estrutura da entrada: considerar melhor caso, pior caso, caso médio?

$\Longrightarrow$ O tempo de cada recursão é abstrato, portanto podemos ignorar constantes e usar *Ordens de Grandeza*.

Exemplo 3

```
(define (sort alon)
  (cond
    [(empty? alon) empty]
    [(cons? alon) (insert (first alon) (sort (rest alon)))]))

(define (insert n alon)
  (cond
    [(empty? alon) (cons n empty)]
    [else (cond
            [(>= n (first alon)) (cons n alon)]
            [(<  n (first alon)) (cons (first alon) (insert n (rest al
```

```
  (sort (list 3 1 2))
= (insert 3 (sort (list 1 2)))
= (insert 3 (insert 1 (sort (list 2))))
= (insert 3 (insert 1 (insert 2 (sort empty))))
= (insert 3 (insert 1 (insert 2 empty)))
= (insert 3 (insert 1 (list 2)))
= (insert 3 (cons 2 (insert 1 empty)))
= (insert 3 (list 2 1))
= (insert 3 (list 2 1))
= (list 3 2 1)
```

Exemplo 4

```
;; maxi : ne-list-of-numbers  ->  number
;; to determine the maximum of a non-empty list of numbers
(define (maxi alon)
  (cond
    [(empty? (rest alon)) (first alon)]
    [else (cond
    [(> (maxi (rest alon)) (first alon)) (maxi (rest alon))]
    [else (first alon)])]))
```

expressão	requer 2 avaliações de
(maxi (list 0 1 2 3))	(maxi (list 1 2 3))
(maxi (list 1 2 3))	(maxi (list 2 3))
(maxi (list 2 3))	(maxi (list 3))

Exemplo 4'

```
;; maxi2 : ne-list-of-numbers  ->  number
;; to determine the maximum of a list of numbers
(define (maxi2 alon)
  (cond
    [(empty? (rest alon)) (first alon)]
    [else (local ((define max-of-rest (maxi2 (rest alon))))
    (cond
      [(> max-of-rest (first alon)) max-of-rest]
      [else (first alon)])])))
```

Ordens de Grandeza

O custo de um programa, com relação ao tamanho da entrada pode ser de ordem

- constante;
- polinomial;
- exponencial

Usando esse custo, pode-se dividir os problemas computáveis em classes que indicam se existe ou não um algoritmo eficiente que possa resolvê-los.

O que vem por aí...

- Estruturas de dados (INA-2)
- Lógica para computação (INT-2)
- Teoria do grafos e análise combinatória (INT-2)
- Classificação e pesquisa de dados (INA-3)
- Computação simbólica e numerica (INT-3)
- Teoria da computação (INT-3)
- Linguagens formais e autômatos (INT-4)
- Complexidade de algoritmos (INT-5)
- Semântica formal (INT-5)