

nome@padrão

nome@padrão é uma forma de fazer uma definição local ao nível de um argumento de uma função.

Exemplos:

A função `fun :: (Int,String) -> (Char,(Int,String))`

pode ser definida, equivalentemente, por:

```
fun (n,(x:xs)) = (x,(n,(x:xs)))
```

ou `fun par@(n,(x:xs)) = (x,par)`

ou `fun (n,(x:xs)) = let par = (n,(x:xs))
in (x,par)`

```
{- Esta função vai retirando os elementos de uma lista até  
encontrar um elemento não positivo -}
```

```
dropWhilePos [] = []  
dropWhilePos lis@(x:xs) | x > 0 = dropWhilePos xs  
| otherwise = lis
```

65

Insertion Sort

Algoritmo:

1. Seleciona-se a cabeça da lista.
2. Ordena-se a cauda da lista.
3. Insere-se a cabeça da lista na cauda ordenada, de forma a que a lista resultante continue ordenada.

```
isort [] = []  
isort (x:xs) = insert x (isort xs)
```

```
insert x [] = [x]  
insert x (y:ys) = if x < y then x:y:ys  
                  else y:(insert x ys)
```

A função **insert** (que faz a inserção ordenada) é o núcleo deste algoritmo.

```
isort [3,5,6,2,7,5,8] => insert 3 (isort [5,6,2,7,5,8])  
=> ... => insert 3 [2,5,5,6,7,8]  
=> ... => [2,3,5,5,6,7,8]
```

67

Algoritmos de Ordenação

A ordenação de um conjunto de valores é um problema muito frequente, e muito útil na organização de informação.

Para o problema de ordenação de uma lista de valores, existem diversos algoritmos:

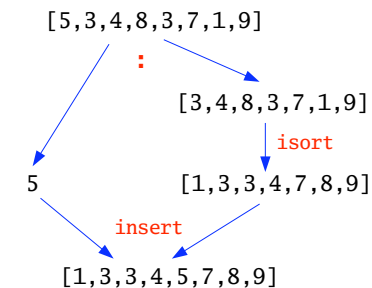
- **Insertion Sort**
- **Quick Sort**
- **Merge Sort**
- ...

Vamos apresentar estes algoritmos, para **ordenar uma lista de valores por ordem crescente**, de acordo com os operadores relacionais **<**, **<=**, **>**, e **>=** (que implicitamente assumimos estarem definidos para os tipos desses valores).

66

Insertion Sort

Exemplo: Esquema do cálculo de `(isort [5,3,4,8,3,1,9])`



68

Quick Sort

Algoritmo:

1. Seleciona-se a cabeça da lista (como *pivot*) e parte-se o resto da lista em duas sublistas: uma com os elementos inferiores ao pivot, e outra com os elementos não inferiores.
2. Estas sublistas são ordenadas.
3. Concatena-se as sublistas ordenadas, de forma adequada, conjuntamente com o pivot.

```
qsort [] = []
qsort (x:xs) = qsort [ y | y <- xs, y < x ]
               ++ [x] ++ qsort [ y | y <- xs, y >= x ]
```

Esta versão do qsort é pouco eficiente ...

Quantas travessias da lista se estão a fazer para partir a lista ?

```
qsort [5,3,4,8,3,7,1,9] =>
... => (qsort [3,4,3,1])++[5]++(qsort [8,7,9])
=> ... => [1,3,3,4] ++ [5] ++ [7,8,9]
=> ... => [1,3,3,4,5,7,8,9]
```

69

Merge Sort

Algoritmo:

1. Parte-se a lista em duas sublistas de tamanho igual (ou quase).
2. Ordenam-se as duas sublistas.
3. Fundem-se as sublistas ordenadas, de forma a que a lista resultante fique ordenada.

```
merge [] l = l
merge l [] = l
merge a@(x:xs) b@(y:ys) | x < y = x:(merge xs b)
                        | otherwise = y:(merge a ys)
```

```
msort [] = []
msort [x] = [x]
msort xs = merge (msort xs1) (msort xs2)
  where
    k = (length xs) `div` 2
    xs1 = take k xs
    xs2 = drop k xs
```

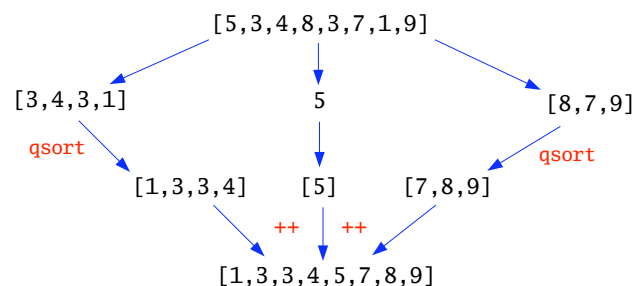
Esta versão do msort é muito pouco eficiente ...

Quantas travessias da lista se está a fazer para partir a lista em duas ?

71

Quick Sort

Exemplo: Esquema do cálculo de (qsort [5,3,4,8,3,1,9])



Uma *versão mais eficiente* (fazendo a partição da lista numa só passagem), pode ser:

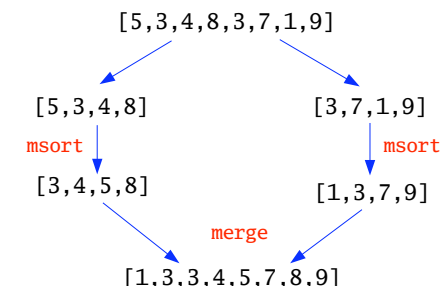
```
parte _ [] = ([],[])
parte x (y:ys) | y < x = (y:as,bs)
                | otherwise = (as,y:bs)
  where (as,bs) = parte x ys
```

```
quicksort [] = []
quicksort (x:xs) = let (l1,l2) = parte x xs
                  in (quicksort l1)++[x]++(quicksort l2)
```

70

Merge Sort

Exemplo: Esquema do cálculo de (msort [5,3,4,8,3,1,9])



Uma *versão mais eficiente* (fazendo a partição da lista numa só passagem), pode ser:

```
split [] = ([],[])
split (x:xs) = let (l,r) = split xs
               in (x:r,l)
```

```
mergesort [] = []
mergesort [x] = [x]
mergesort l = merge (mergesort l1) (mergesort l2)
  where (l1,l2) = split l
```

72

Acumuladores

Considere a definição da função **factorial**.

```
fact 0 = 1
fact n | n>0 = n * fact (n-1)
```

O cálculo da factorial de um número positivo n é feito multiplicando n pelo factorial de $(n-1)$.

A multiplicação fica *em suspenso* até que o valor de $\text{fact } (n-1)$ seja sintetizado.

```
fact 3 => 3*(fact 2) => 3*(2*(fact 1)) => 3*(2*(1*(fact 0)))
=> 3*(2*(1*1)) => 6
```

Uma outra estratégia para resolver o mesmo problema, consiste em definir uma função auxiliar com um parametro extra que serve para *ir guardando os resultados parciais* – a este parametro extra chama-se **acumulador**.

```
fact n | n >= 0 = factAc 1 n
  where factAc ac 0 = ac
        factAc ac n = factAc (ac*n) (n-1)
```

```
fact 3 => factAc 1 3 => factAc (1*3) 2 => factAc (1*3*2) 1
=> factAc (1*2*3*1) 0 => 1*2*3*1 => 6
```

73

Considere a função que inverte uma lista.

```
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

```
reverse [1,2,3] => (reverse [2,3])++[1] => ((reverse [3])++[2])++[1]
=> (((reverse [])++[3])++[2])++[1] => (([]++[3])++[2])++[1]
=> ([3]++[2])++[1] => (3:([2]++[1]))++[1] => (3:[2])++[1]
=> 3:([2]++[1]) => 3:(2:([1]++[1])) => 3:2:[1] = [3,2,1]
```

Este é um exemplo típico de uma função que implementada com um acumulador é muito **mais eficiente**.

```
reverse l = revAc [] l
  where revAc ac [] = ac
        revAc ac (x:xs) = revAc (x:ac) xs
```

```
reverse [1,2,3] => revAc [] [1,2,3] => revAc [1] [2,3]
=> revAc [2,1] [3] => revAc [3,2,1] [] => [3,2,1]
```

75

Dependendo do problema a resolver, o uso de acumuladores pode ou não trazer vantagens.

Por vezes, pode ser a forma mais natural de resolver um problema.

Exemplo:

Considere as duas versões da função que faz o cálculo do valor máximo de uma lista.

Qual lhe parece mais natural?

```
maximum [x] = x
maximum (x:y:xs) | x > y      = maximum (x:xs)
                  | otherwise = maximum (y:xs)
```

```
maximo (x:xs) = maxAc x xs
  where maxAc ac [] = ac
        maxAc ac (y:ys) = if y>ac then maxAc y ys
                          else maxAc ac ys
```

Em **maximo** o acumulador guarda o valor máximo encontrado até ao momento.

Em **maximum** a cabeça da lista está a funcionar como acumulador.

74

Funções de Ordem Superior

Em Haskell, as funções são entidades de primeira ordem, isto é, as **funções** podem **ser passadas como parametro** e/ou **devolvidas como resultado** de outras funções

Exemplo: A função **app** tem como argumento uma função **f** de tipo **a->b**.

```
app :: (a->b) -> (a,a) -> (b,b)    app fact (5,4) => (120,24)
app f (x,y) = (f x, f y)           app chr (65,70) => ('A','F')
```

Exemplo:

A função **mult** pode ser entendida como tendo **dois argumentos** de tipo **Int** e devolvendo um valor do tipo **Int**. Mas, na realidade, **mult** é uma função que recebe **um argumento** do tipo **Int** e devolve uma função de tipo **Int->Int**.

```
mult :: Int -> Int -> Int  == Int -> (Int -> Int)
mult x y = x * y
```

Em Haskell, todas as funções são unárias !

```
mult 2 5 == (mult 2) 5 :: Int
          (mult 2) :: Int -> Int
```

76