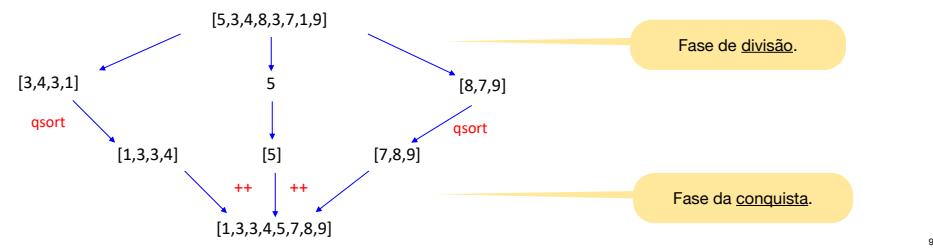


## Quick sort

Este algoritmo segue uma estratégia chamada de “divisão e conquista”.

- Quando a lista não é vazia, **seleciona-se a cabeça** da lista e **parte-se a cauda em duas listas**:
  - uma lista com os elementos que são mais pequenos do que a cabeça,
  - e outra lista com os restantes elementos (isto é, os que são maiores ou iguais à cabeça)
- Depois **ordenam-se estas listas mais pequenas** pelo mesmo método.
- Finalmente **concatenam-se as listas ordenadas e a cabeça**, de forma a que a lista final fique ordenada.



93

## Quick sort

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = (qsort l1) ++ [x] ++ (qsort l2)
  where (l1,l2) = parte x xs
```

A função `parte` pode ser feita usando a técnica de *tuppling*

```
parte :: (Ord a) => a -> [a] -> ([a],[a])
parte [] = ([],[])
parte x (y:ys) | y < x = (y:as, bs)
  | otherwise = (as, y:bs)
  where (as,bs) = parte x ys
```

```
qsort [4,7,2] = (qsort ...) ++ [4] ++ (qsort ...) = (qsort [2]) ++ [4] ++ (qsort [7])
= ... = [2] ++ [4] ++ [7] = [2,4,7]

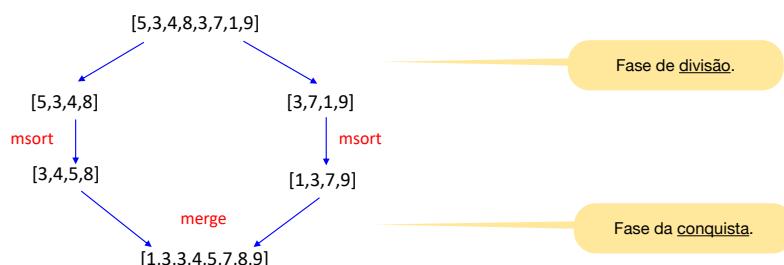
parte 4 [7,2] = (..., 7:...)
parte 4 [2] = (2:..., ...)
parte 4 [] = ([],[])
```

94

## Merge sort

Este algoritmo segue uma estratégia de “divisão e conquista”.

- Quando a lista tem mais do que um elemento, **parte-se a lista em duas listas de tamanho aproximadamente igual** (podem diferir em uma unidade).
- Depois **ordenam-se estas listas mais pequenas** pelo mesmo método.
- Finalmente faz-se a **fusão das duas listas ordenadas** de forma a que a lista final fique ordenada.



95

## Merge sort

Começemos pela função `merge` que faz a **fusão de duas listas ordenadas**.

```
merge :: (Ord a) => [a] -> [a] -> [a]
merge [] l = l
merge l [] = l
merge (x:xs) (y:ys) | x < y = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
```

A função de ordenação pode definir-se assim:

```
msort :: (Ord a) => [a] -> [a]
msort [] = []
msort [x] = [x]
msort l = merge (msort (l1)) (msort (l2))
  where n = (length l) `div` 2
        (l1,l2) = splitAt n l

msort [4,7,2] = merge (msort [4]) (msort [7,2]) = ... = merge [4] [2,7]
= 2 : merge [4] [7]
= 2 : 4 : merge [] [7]
= 2 : 4 : [7] = [2,4,7]

porque splitAt 1 [4,7,2] = ([4], [7,2])
```

96

## Merge sort

Podemos definir a função `msort` de outro modo:

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

`split` parte a lista numa só travessia. A lista está ser partida de maneira diferente, mas isso não tem interferência no algoritmo.

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

split :: [a] -> ([a],[a])
split [] = ([],[])
split [x] = ([x],[])
split (x:y:t) = (x:l1, y:l2)
  where (l1,l2) = split t

msort :: (Ord a) => [a] -> [a]
msort [] = []
msort [x] = [x]
msort l = merge (msort l1) (msort l2)
  where (l1,l2) = split l
```

97

## Funções com parâmetro de acumulação

- A ideia que está na base destas funções é que elas vão ter um parâmetro extra (o **acumulador**) onde a resposta vai sendo construída e gravada à medida que a recursão progride.
- O acumulador vai sendo actualizado e passado como parâmetro nas sucessivas chamadas da função.
- Uma vez que o acumulador vai guardando a resposta da função, o seu tipo deve ser igual ao tipo do resultado da função.

**Exemplo:** A função que inverte uma lista.

A função inverte chama uma função auxiliar `inverteAc` com um parâmetro de acumulação e inicializa o acumulador.

```
inverte :: [a] -> [a]
inverte l = inverteAc l []
  where inverteAc [] ac = ac
        inverteAc (x:xs) ac = inverteAc xs (x:ac)
```

A chamada recursiva é feita actualizando o acumulador.

O acumulador é inicialmente a lista vazia.

Quando a lista é vazia o acumulador tem a solução completa.

98

## Funções com parâmetro de acumulação

```
inverte :: [a] -> [a]
inverte l = inverteAc l []
  where inverteAc [] ac = ac
        inverteAc (x:xs) ac = inverteAc xs (x:ac)
```

```
inverte [1,2,3] = inverteAc [1,2,3] []
  = inverteAc [2,3] [1]
  = inverteAc [3] [2,1]
  = inverteAc [] [3,2,1]
  = [3,2,1]
```

A solução está a ser construída no acumulador.

Esta versão é bastante mais eficiente que a função `reverse` anteriormente definida (porque usa o `++` que tem que atravessar a primeira lista).

```
reverse :: [a] -> [a]
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]
```

99

## Funções com parâmetro de acumulação

Podemos sistematizar as seguintes regras para definir funções usando esta técnica:

1. Colocar o acumulador como um parâmetro extra.
2. O acumulador deve ser do mesmo tipo que o do resultado da função.
3. Devolver o acumulador no acaso de paragem da função.
4. Actualizar o acumulador na chamada recursiva da função.
5. A função principal (sem acumulador) chama a função com parâmetro de acumulação, inicializando o acumulador.

**Exemplo:** O somatório de uma lista de números.

```
somatorio :: Num a => [a] -> a
somatorio l = sumAc l 0
  where sumAc :: Num a => [a] -> a -> a
        sumAc [] n = n
        sumAc (x:xs) n = sumAc xs (x+n)
```

```
somatorio [1,2,3]
  = sumAc [1,2,3] 0
  = sumAc [2,3] (1+0)
  = sumAc [3] (2+1+0)
  = sumAc [] (3+2+1+0)
  = 6
```

100

## Funções com parâmetro de acumulação

Exemplo: O máximo de uma lista não vazia.

```
maximo :: Ord a => [a] -> a
maximo (x:xs) = maxAc xs x
  where maxAc :: Ord a => [a] -> a -> a
    maxAc [] n = n
    maxAc (x:xs) n = if x > n then maxAc xs x
                      else maxAc xs n
```

```
maximo [2,7,3,9,4] = maxAc [7,3,9,4] 2
  = maxAc [3,9,4] 7
  = maxAc [9,4] 7
  = maxAc [4] 9
  = maxAc [] 9
  = 9
```

## Funções com parâmetro de acumulação

Exemplo: A função factorial.

```
factorial :: Integer -> Integer
factorial n = factAc n 1
  where factAc :: Integer -> Integer -> Integer
    factAc 0 x = x
    factAc n x | n>0 = factAc (n-1) (n*x)
```

```
factorial 5 = factAc 5 1
  = factAc 4 (5*1)
  = factAc 3 (4*5*1)
  = factAc 2 (3*4*5*1)
  = factAc 1 (2*3*4*5*1)
  = factAc 0 (1*2*3*4*5*1)
  = 120
```

## Funções com parâmetro de acumulação

Exemplo: A função `stringToInt :: String -> Int` que converte uma string (representando um número inteiro positivo) num valor inteiro.

```
stringToInt "5247" = 5247

import Data.Char

stringToInt :: String -> Int
stringToInt (x:xs) = aux xs (digitToInt x)
  where aux :: String -> Int -> Int
    aux (h:t) ac = aux t (ac*10 + (digitToInt h))
    aux [] ac = ac
```

```
stringToInt "5247" = aux "247" 5
  = aux "47" (50+2)
  = aux "7" (520+4)
  = aux "" (5240+7)
  = 5247
```

## Listas por compreensão

Na matemática é costume definir conjuntos por compreensão à custa de outros conjuntos.

$\{ 2x \mid x \in \{10,3,7,2\} \}$

O conjunto {20,6,14,4}.

$\{ n \mid n \in \{4, -5, 8, 20, -7, 1\} \wedge 0 \leq n \leq 10 \}$

O conjunto {4,8,1}.

Em Haskell podem definir-se [listas por compreensão](#), de modo semelhante, construindo novas listas à custa de outras listas.

`[ 2*x | x <- [10,3,7,2] ]`

A lista [20,6,14,4].

`[ n | n <- [4,-5,8,20,-7,1], 0<=n, n<=10 ]`

A lista [4,8,1].