Tupling *(calcular vários resultados numa só travessia da lista)*

- Numa só travessia podemos ir somando os valores das respectivas componentes, mantendo um par que vamos construindo.

```
somas :: [(Int,Int)] -> (Int,Int)
somas [] = (0,0)
somas ((x,y):t) = (x + fst (somas t), y + snd (somas t))
```

Note que `(soma t)` devolve um par. Daí o uso das funções `fst` e `snd`.

Pode parecer que `(soma t)` está a ser calculada duas vezes, mas isso não é verdade. `(soma t)` só é calculado uma vez, já que o valor dos identificador numa é alterado.

- Podemos fazer uma declaração local para tornar o código mais fácil de ler

```
somas :: [(Int,Int)] -> (Int,Int)
somas [] = (0,0)
somas ((x,y):t) = (x + a, y + b)
   where (a,b) = somas t
```

Estamos aqui a usar o padrão `(a,b)` para extrair as componentes do par devolvido pela por `(somas t)`.

Tupling *(calcular vários resultados numa só travessia da lista)*

```
somas :: [(Int,Int)] -> (Int,Int)
somas [] = (0,0)
somas ((x,y):t) = (x + fst (somas t), y + snd (somas t))
```

```
somas [(7,8),(1,2)] = (7+ fst (somas [(1,2)]), 8+ snd (somas [(1,2)]))
                    = (7+ fst (1+ fst (somas []), 2+ fst (somas [])) ,
                       8+ snd (1+ fst (somas []), 2+ snd (somas [])) )
                    = (7+ fst (1+ fst (0,0), 2+ fst (0,0)) ,
                       8+ snd (1+ fst (0,0), 2+ snd (0,0)) )
                    = (7+ fst (1+0, 2+0) , 8+ snd (1+0, 2+0) )
                    = (7+1+0 , 8+2+0)
                    = (8, 10)
```

Tupling *(calcular vários resultados numa só travessia da lista)*

```
somas :: [(Int,Int)] -> (Int,Int)
somas [] = (0,0)
somas ((x,y):t) = (x+a, y+b)
   where (a,b) = somas t
```

```
somas [(7,8),(1,2)] = (7+ …, 8+ …) = (7+1+0, 8+2+0) = (8,10)
    somas [(1,2)] = (1+ …, 2+ …)   = (1+0, 2+0)
        somas [] = (0,0)
```

Funções recursivas sobre listas

- **`zip`** emparelha duas listas.

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _ = []
```

- **`unzip`** separa uma lista de pares em duas listas.

```
unzip :: [(a,b)] -> ([a],[b])
unzip [] = ([],[])
unzip ((x,y):t) = (x:e, y:d)
  where (e,d) = unzip t
```

```
unzip [(1,True),(2,False),(3,True)] = (1:… , True:…) = (1:2:3:[], True:False:True:[])
  unzip [(2,False),(3,True)] = (2:… , False:…)       = (2:3:[], False:True:[])
    unzip [(3,True)] = (3:… , True:…)                = (3:[], True:[])
      unzip [] = ([],[])
```

Funções recursivas sobre listas

- **take** dá os n primeiros elementos de uma lista.

```
take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

```
take 2 [7,5,3] = 7 : take 1 [5,3]
               = 7 : 5 : take 0 [3]
               = 7 : 5 : []
               = [7,5]
```

```
take 2 [7] = 7 : take 1 []
           = 7 : []
           = [7]
```

Funções recursivas sobre listas

- **drop** retira os n primeiros elementos de uma lista.

```
drop :: Int -> [a] -> [a]
drop n xs | n <= 0  = xs
drop _ [] = []
drop n (_:xs) = drop (n-1) xs
```

```
drop 2 [7,5,3] = drop 1 [5,3]
               = drop 0 [3]
               = [3]
```

```
drop 2 [7] = drop 1 []
           = []
```

Tuppling

- **splitAt** parte uma lista em duas da seguinte forma

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n l = (take n l, drop n l)
```

Esta função recorre às funções `take` e `drop` como funções auxiliares. Há no entanto algum desperdício de trabalho nesta implementação, porque se está a percorrer a lista até à posição `n` duas vezes sem necessidade.

Podemos definir a função assim

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n l | n <= 0  = ([],l)
splitAt _ [] = ([],[])
splitAt n (x:xs) = (x:l1, l2)
    where (l1,l2) = splitAt (n-1) xs
```

```
splitAt 2 [7,8,9,0,1] = (7:… , …)                            = (7:8:[] , [9,0,1])
   splitAt 1 [8,9,0,1] = (8:… , …)        = (8:[] , [9,0,1])
      splitAt 0 [9,0,1] = ([],[9,0,1])
```

Lazy evaluation

- O Haskell faz o cálculo do valor de uma expressão usando as equações que definem as funções como regras de cálculo.
- Cada passo do processo de cálculo costuma chamar-se de redução.
- Cada redução resulta de substituir a instância do lado esquerdo da equação pelo respectivo lado direito.
- A estratégia de redução usada para o cálculo das expressões é uma característica essencial de uma linguagem funcional.

Exemplo: considere as seguintes funções

```
dobro x = x + x
snd (x,y) = y
```

Como é que a expressão `dobro (snd (3,7))` é calculada?

Há duas possibilidades:

```
dobro (snd (3,7)) = dobro 7 = 7 + 7 = 14
```

ou

```
dobro (snd (3,7)) = snd (3,7) + snd (3,7) = 7 + 7 = 14
```

Lazy evaluation

- O Haskell usa como estratégia de redução a *lazy evaluation* (também chamada de *call-by-name*).
- A lazy evaluation caracteriza-se por aplicar as funções sem fazer o cálculo prévio dos seus argumentos.
- A sequência de redução que o Haskell faz no cálculo da expressão `dobro (snd (3,7))` é

```
dobro (snd (3,7)) = snd (3,7) + snd (3,7) = 7 + 7 = 14
```

- Com a *lazy evaluation* os argumentos das funções só são calculados se o seu valor fôr mesmo necessário.

```
snd (sqrt (20^3+ (45/23)^10), 1) = 1
```

- A *lazy evaluation* faz do Haskell uma linguagem não estrita. Isto é, uma função aplicada a um valor indefinido pode ter em Haskell um valor bem definido.

```
snd (5 `div` 0, 1) = 1
```

- A *lazy evaluation* também vai permitir ao Haskell lidar com estruturas de dados infinitas.

Algoritmos de ordenação

- A ordenação de um conjunto de valores é um problema muito frequente e muito útil na organização de informação.
- Para resolver o problema de ordenação de uma lista existem muitos algoritmos. Vamos estudar três desses algoritmos:
  - *Insertion sort*
  - *Quick sort*
  - *Merge sort*
- Vamos apresentar esses algoritmos para ordenar uma lista por ordem crescente.

Insertion sort

Este algoritmo apoia-se numa função auxiliar que insere um elemento numa lista já ordenada.

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

```
insert 7 [2,5,9]
     = 2 : insert 7 [5,9]
     = 2 : 5 : insert 7 [9]
     = 2:5:7:9:[]
     = [2,5,7,9]
```

A função de ordenação da lista define-se por casos:

```
isort :: (Ord a) => [a] -> [a]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

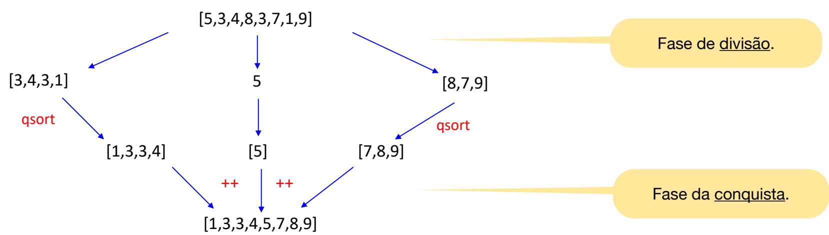
Se a lista não é vazia, insere o elemento da cabeça da lista na cauda previamente ordenada pelo mesmo método.

```
isort [4,7,2] = insert 4 (isort [7,2])
              = insert 4 (insert 7 (isort [2]))
              = insert 4 (insert 7 (insert 2 (isort [])))
              = insert 4 (insert 7 (insert 2 []))
              = insert 4 (insert 7 [2]) = …
              = insert 4 [2,7] = … = [2,4,7]
```

Quick sort

Este algoritmo segue uma estratégia chamada de "divisão e conquista".

- Quando a lista não é vazia, selecciona-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.



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 *tupling*

```
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
```

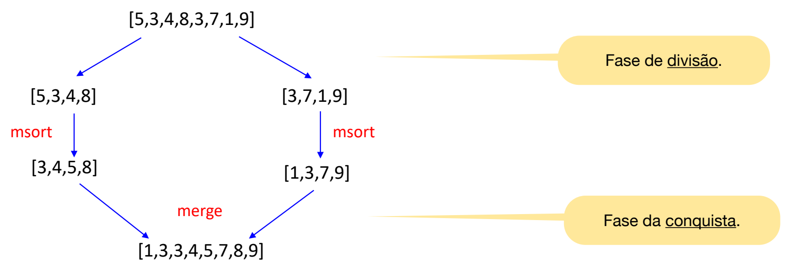
```
qsort [4,7,2] = (qsort …) ++ [4] ++ (qsort …) = (qsort [2]) ++ [4] ++ (qsort [7])
                                              = … = [2] ++ [4] ++ [7] = [2,4,7]

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

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.



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 l1)
   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])
```

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.

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

```
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 l1)
   where (l1,l2) = split l
```