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.

```
take 7 [5,10..] = [5,10,15,20,25,30,35]
```

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)
```

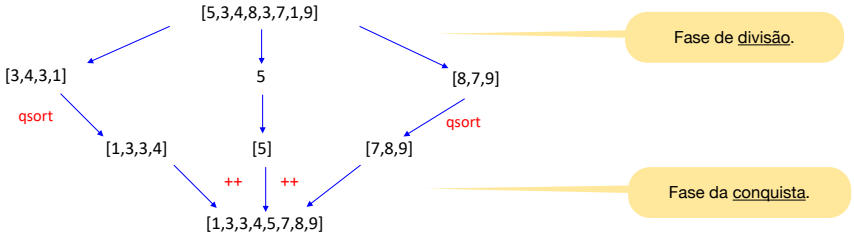
Se a lista não é vazia, insere o elemento da cabeça da lista na cauda previamente ordenada pelo mesmo método.

```
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 *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
```

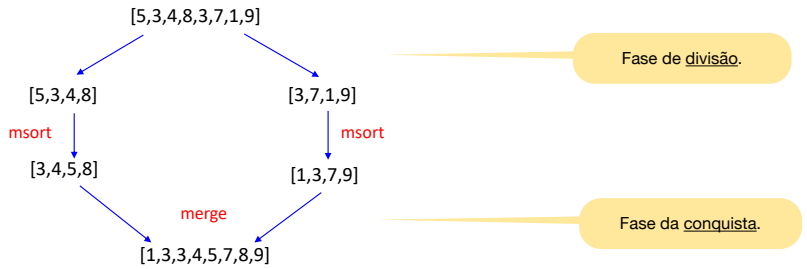
```
qsort [4,7,2] = (qsort …) ++ [4] ++ (qsort …) = (qsort [2]) ++ [4] ++ (qsort [7])
                                              = … = [2] ++ [4] ++ [7] = [2,4,7]

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


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.

O acumulador é inicialmente a lista vazia.

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

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

A chamada recursiva é feita actualizando o acumulador.