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)
```

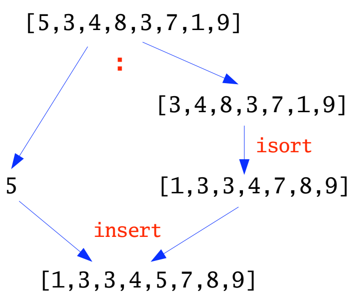
```
insert x [] = [x]
insert x (y:ys) = if x<y then x:y:ys
                         else y:(insert x ys)
```

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]
```


Insertion Sort

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



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 adquada, conjuntamente com o pivot.

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

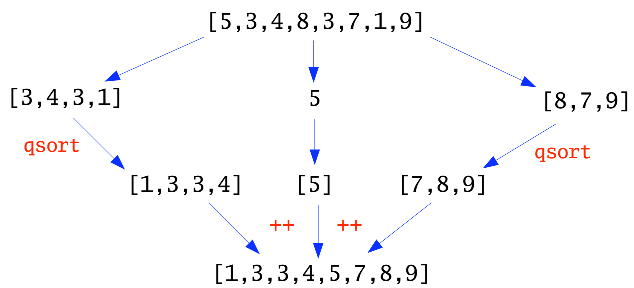
Esta versão do `qsort` é pouco eficiente ...

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]
```


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)
```

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)
```

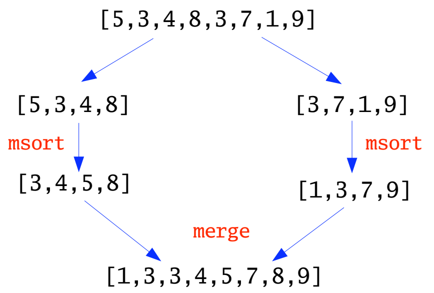
```
msort []  = []
msort [x] = [x]
msort xs  = merge (msort xs1) (msort xs2)
   where                           -- pouco eficiente ...
         k   = (lenght xs) `div` 2
         xs1 = take k xs
         xs2 = drop k xs
```

Esta versão do `msort` é muito pouco eficiente ...

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

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
```


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 `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
```


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:ys)
                 | 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.

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]  ⇒ (3:[2])++[1]
          ⇒ 3:([2]++[1]) ⇒ 3:(2:([]++[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]
```


Sequência de Fibonacci

O n-ésimo número da sequência de Fibonacci define-se matematicamente por

$$\begin{array}{ll} fib\ n = 0 & ,\ se\ \ n = 0 \\ fib\ n = 1 & ,\ se\ \ n = 1 \\ fib\ n = fib\ (n\text{-}2) + fib\ (n\text{-}1) & ,\ se\ \ n \geq 2 \end{array}$$

```
fib 0 = 0
fib 1 = 1
fib n | n>=2  = fib (n-2) + fib (n-1)
```

O cálculo do fib de um número pode envolver o cálculo do fib de números mais pequenos, repetidas vezes.

```
fib 5 ⇒  (fib 3)+(fib 4)  ⇒  ((fib 1)+(fib 2))+((fib 2)+(fib 3))
 ⇒ (1+((fib 0)+(fib 1)))+((fib 2)+(fib 3)) ⇒  ... ⇒  ... ⇒ 5
```

A sequência de Fibonacci pode ser definida por

```
seqFibonacci = [ fib n | n <- [0,1..]  ]
```


Uma versão mais eficiente dos números de Fibonacci utiliza um parametro de acumulação.

Neste caso o acumulador é um par que regista os dois últimos números de Fibonacci calculados até ao momento.

```
fib n = fibAc (0,1) n
  where fibAc (a,b) 0 = a
        fibAc (a,b) 1 = b
        fibAc (a,b) (n+1) = fib (b,a+b) n
```

```
fib 5  ⇒  fibAc (0,1) 5  ⇒  fibAc (1,1) 4  ⇒  fibAc (1,2) 3
       ⇒  fibAc (2,3) 2  ⇒  fibAc (3,5) 1  ⇒  5
```

A sequência de Fibonnacci pode ser definida por

```
seqFib = 0 : 1 : [ a+b | (a,b) <- zip seqFib (tail seqFib) ]
```

Note que é a lazy evaluation que faz com que este género de definição seja possível.

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 f (x,y) = (f x, f y)
```

```
app fact (5,4) ⇒ (120,24)

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 a funções são unárias !

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

Assim, `mult` pode ser usada para *gerar novas funções*.

Exemplo:

```
dobro = mult 2

triplo = mult 3
```

Qual é o seu tipo ?

Os operadores infixos também podem ser usados da mesma forma, isto é, aplicados a apenas um argumento, gerando assim uma nova função.

Exemplo:

```
(+) :: Integer -> Integer -> Integer
(<=) :: Integer -> Integer -> Bool
(*) :: Double -> Double -> Double
```

`(5+)` `(+) 5 :: Integer -> Integer`

`(0<=)` Qual é o tipo destas funções ?

`(3*)` Qual o valor das expressões:

```
(0<=) 8
(3*) 5.7
```


map

Considere as seguintes funções:

```
distancias :: [Ponto] -> [Float]
distancias [] = []
distancias (p:ps) = (distOrigem p) : (distancias ps)
```

```
minusculas :: String -> String
minusculas [] = []
minusculas (c:cs) = toLower c : minusculas cs
```

```
triplica :: [Double] -> [Double]
triplica [] = []
triplica (x:xs) = (3*x) : triplica xs
```

```
factoriais :: [Integer] -> [Integer]
factoriais [] = []
factoriais (n:ns) = fact n : factoriais ns
```

Todas estas funções têm um *padrão de computação* comum:

aplicam uma função a cada elemento de uma lista, gerando deste modo uma nova lista.

map

Podemos definir uma função de ordem superior que aplica uma função ao longo de uma lista:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

Note que `(map f lista)` é equivalente a `[ f x | x <- lista ]`

Podemos definir as funções do slide anterior à custa da função `map`, fazendo:

```
distancias lp = map distOrigem lp
minusculas s = map toLower s
triplica xs = map (3*) xs
factoriais ns = map fact ns
```

Ou então,

```
distancias = map distOrigem
minusculas = map toLower
triplica = map (3*)
factoriais = map fact
```

Porquê ?

filter

Considere as seguites funções:

```
aprov :: [Int] -> [Int]
aprov [] = []
aprov (x:xs) = if (10<=x) then x:(aprov xs)
                          else (aprov xs)
```

```
filtraDigitos :: String -> String
filtraDigitos [] = []
filtraDigitos (c:cs)
     | isDigit c = c:(filtraDigitos cs)
     | otherwise = filtraDigitos cs
```

```
primQuad :: [Ponto] -> [Ponto]
primQuad [] = []
primQuad ((x,y):ps) | x>0 && y>0 = (x,y):(primQuad ps)
                    | otherwise  = primQuad ps
```

Todas estas funções têm um *padrão de computação* comum:

dada uma lista, geram uma nova lista com os elementos da lista que satisfazem um determinado predicado.

filter

filter é uma função de ordem superior que filtra os elementos de uma lista que verificam um dado predicado (i.e. mantém os elementos da lista para os quais o predicado é verdadeiro).

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
     | (p x)     = x : (filter p xs)
     | otherwise = filter p xs
```

Note que `(filter p lista)` é equivalente a `[ x | x <- lista , p x ]`

Podemos definir as funções do slide anterior à custa da função filter, fazendo:

```
aprov xs = filter (10<=) xs
```

```
filtraDigitos s = filter isDigit s
```

```
primQuad ps = filter aux ps
  where  aux (x,y) = 0<x && 0<y
```

Ou então,

```
aprov = filter (10<=)
```

```
filtraDigitos = filter isDigit
```

```
primQuad = filter aux
  where  aux (x,y) = 0<x && 0<y
```


Funções anónimas

Em Haskell, é possível definir novas funções através de *abstrações lambda* (λ) da forma:

\x -> e representando uma função com argumento formal x e corpo da função e (a notação é inspirada no λ-*calculus* aonde isto se escreve $\lambda x.e$)

Exemplos:

```
> (\x -> x+x) 5
10
```

```
> (\y -> y*3) 4
12
```

```
> (\x -> x:x^2:x^3:[]) 2
[2,4,8]
```

Funções com mais do que um argumento podem ser definidas de forma *abreviada* por:

\p1 ... pn -> e Além disso, os argumentos p1 ... pn podem ser padrões.

Exemplos:

```
> (\x y -> x+y) 5 3
8
```

```
> (\(x:xs) y -> y:xs) [3,4,5,2] 7
[7,4,5,2]
```

```
> (\(x1,y1) (x2,y2) -> (x1*x2,y1*y2)) (0,3) (5,2)
(0,6)
```

Note que: `\x y -> x+y` ≡ `\x -> (\y -> x+y)` Justifique com base no tipo.

Como ao definir estas funções não lhes associamos um nome, elas dizem-se **anónimas**.

Funções anónimas

É possível utilizar funções anónimas na definição de outras funções.

Exemplos:

```
dobro = \x->x+x
```

```
cauda = \(_:xs) -> xs
```

```
> dobro 5
10
> cauda [9,3,4,5]
[3,4,5]
```

As funções anónimas são úteis para evitar a declaração de funções auxiliares.

Exemplos:

```
trocaPares xs = map troca xs
  where troca (x,y) = (y,x)
```

```
trocaPares xs = map (\(x,y)->(y,x)) xs
```

```
primQuad = filter (\(x,y) -> 0<x && 0<y)
```

Os operadores infixos aplicados apenas a um argumento são uma forma abreviada de escrever funções anónimas.

Exemplos:

`(+y)` ≡ `\x -> x+y`

`(x+)` ≡ `\y -> x+y`

`(*5)` ≡ `\x -> x*5`

foldr

Considere as seguintes funções:

```
sum [] = 0
sum (x:xs) = x + (sum xs)
```

`sum [3,5,8]` ⇒ `3 + (5 + (8+0))`

```
product [] = 1
product (x:xs) = x * (product xs)
```

```
and [] =  True
and (b:bs) = b && (and bs)
```

```
concat [] = []
concat (l:ls) = l ++ (concat ls)
```

Todas estas funções têm um *padrão de computação* comum:

aplicar um operador binário ao primeiro elemento da lista e ao resultado de aplicar a função ao resto da lista.

O que se está a fazer é a extensão de uma operação binária a uma lista de operandos.

foldr

Podemos capturar este padrão de computação fornecendo à função `foldr` o operador binário e o resultado a devolver para a lista vazia.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Note que `(foldr f z [x1,...,xn])` é igual a `(f x1 (... (f xn z)...))`
ou seja, `(x1 `f` (x2 `f` (... (xn `f` z)...)))` *(associa à direita)*

Podemos definir as funções do slide anterior à custa da função `foldr`, fazendo:

```
sum xs = foldr (+) 0 xs
```

```
product xs = foldr (*) 1 xs
```

```
and bs = foldr (&&) True bs
```

```
concat ls = foldr (++) [] ls
```

Exemplos:

```
(product [4,3,5]) ⇒ 4 * (3 * (5 * 1)) ⇒ 60
(concat [[3,4,5],[2,1],[7,8]]) ⇒ [3,4,5] ++ ([2,1] ++ ([7,8]++[]))
                               ⇒ [3,4,5,2,1,7,8]
```


foldl

Podemos usar um padrão de computação semelhante ao do `foldr`, mas *associando à esquerda*, através da função `foldl`.

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Note que `(foldl f z [x1,...,xn])` é igual a `(f (...(f z x1) ...) xn)`
ou seja, `((...((z `f` x1) `f` x2)...) `f` xn)` *(associa à esquerda)*

Exemplos:

```
sum xs = foldl (+) 0 xs
```

```
concat ls = foldl (++) [] ls
```

```
reverse xs = foldl (\t h -> h:t) [] xs
```

```
sum [1,2,3] ⇒ ((0 + 1) + 2) + 3 ⇒ 6

concat [[2,3],[8,4,7],[1]] ⇒ (([]++[2,3]) ++ [8,4,7]) ++ [1]
                           ⇒ [2,3,8,4,7,1]

reverse [3,4] ⇒ ((\t h -> h:t) ((\t h -> h:t) [] 3) 4)
              ⇒ 4 : ((\t h -> h:t) [] 3) ⇒ 4:3:[] ⇒ [4,3]
```


foldr *vs* foldl

Note que `(foldr f z xs)` e `(foldl f z xs)` só darão o mesmo resultado se a função `f` for comutativa e associativa, caso contrário dão resultados distintos.

Exemplo:

```
foldr (-) 8 [4,7,3,5] ⇒ 4 - (7 - (3 - (5 - 8))) ⇒ 3

foldl (-) 8 [4,7,3,5] ⇒ (((8 - 4) - 7) - 3) - 5 ⇒ -11
```

As funções `foldr` e `foldl` estão formemente relacionadas com as estratégias para contruir funções recursivas sobre listas que vimos atrás.

`foldr` está relacionada com a *recursividade primitiva.*

`foldl` está relacionada com o *uso de acumuladores.*

Exercício: Considere as funções

```
sumR xs = foldr (+) 0 xs
```

```
sumL xs = foldl (+) 0 xs
```

Escreva a cadeia de redução das expressões `(sumR [1,2,3])` e `(sumL [1,2,3])` e compare com o funcionamento da função somatório definida sem e com e acumuladores.

Outras funções de ordem superior

Composição de funções

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

Trocar a ordem dos argumentos

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

Obter a versão *curried* de uma função

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f x y = f (x,y)
```

Obter a versão *uncurried* de uma função

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x,y) = f x y
```

Exemplos:

```
sextuplo = dobro . triplo
```

```
reverse xs = foldl (flip (:)) [] xs
```

```
quocientes pares = map (uncurry div) pares
```

```
sextuplo 5 ⇒ dobro (triplo 5) ⇒ dobro 15 ⇒ 30

quocientes [(3,4),(23,5),(7,3)] ⇒ [0,4,2]
```