Considere a função `zip` já definida no Perlude:

```
zip [] [] = []
zip [] (y:ys) = []
zip (x:xs) [] = []
zip (x:xs) (y:ys) = (x,y) : (zip xs ys)
```

Qual o seu tipo ?   É total ou parcial ?
Podemos trocar a ordem das equações ?
Podemos dispensar alguma equação ?
Será que podemos definir `zip` com menos equações ?

Exercícios:

- Indique todos os passos de redução envolvidos no cálculo da expressão:

  `zip [1,2] “LMCC”`

- Defina a função que faz o “zip” de 3 listas.
- Defina a função `unzip :: [(a,b)] -> ([a],[b])`

Padrões sobre números naturais.

O Haskell aceita como um padrão sobre números naturais, expressões da forma:

(*variável* + *número_natural*)

Exemplos:

```
fact 0 = 1
fact (n+1) = (n+1) * (fact n)
```

```
decTres (x+3) = x
```

```
> fact 4
24
> fact (-2)
*** Exception: Non-exhaustive patterns in function fact
```

```
> decTres 5
2
> decTres 10
7
> decTres 2
*** Exception: Non-exhaustive ...
```

Atenção: expressões como

`(n*5), (x-4) ou (2+n)`

não são padrões !

Mais alguma funções sobre listas pré-definidas no `Prelude`.

```
(x:_) !! 0   = x
(_:xs) !! (n+1) = xs !! n
```

```
init [x] = []
init (x:xs) = x : init xs
```

O que fazem estas funções ?

Qual o seu tipo ?

Estas funções serão totais ?

Trocando a ordem das equações, será que obtemos a mesma função ?

As funções `take` e `drop` estão pré-definidas no `Prelude` da seguinte forma:

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

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

Estas funções serão totais ?

Trocando a ordem das equações, será que obtemos a mesma função ?

Defina funções equivalentes utilizando padrões de números naturais.

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
```


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 implicítamente assumimos estarem definidos para os tipos desses valores).

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)
```

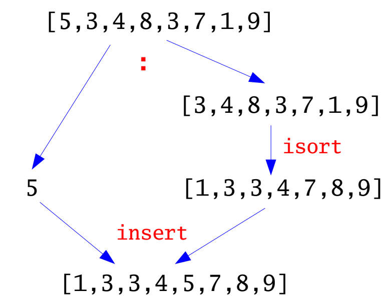
```
insert x [] = [x]
insert x (y:ys) = if x<y then x:y:ys
                         else y:(insert x ys)
```

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 ]
```

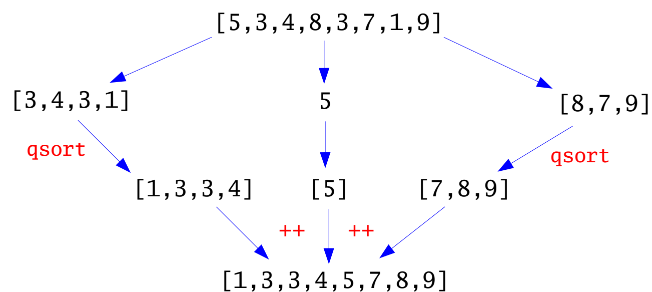
Esta versão do `qsort` é pouco eficiente ...

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)
```

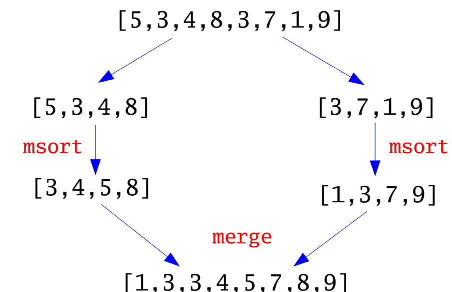
```
msort []  = []
msort [x] = [x]
msort xs  = merge (msort xs1) (msort xs2)
   where                         -- pouco eficiente ...
         k   = (lenght xs) `div` 2
         xs1 = take k xs
         xs2 = drop k xs
```

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
```