```haskell
-- * ============================================= *
-- *                                               *
-- *                Arquivo BONUS                  *
-- *   Codigos exemplos: exercicios de aprendizagem   *
-- *   -------------------------------------------   *
-- *                                               *
-- * Parte integrante do livro: Algoritmos Funcionais *
-- * Copyright: Jose Augusto N. G. Manzano          *
-- *                                               *
-- * A permissao de uso deste material esta vinculado *
-- * a aquisicao do livro correspondente.           *
-- *                                               *
-- * Scripts dos codigos para a linguagem Haskell     *
-- * usados como ilustracao para a fixacao dos con-   *
-- * ceitos logicos apresentados no livro Algoritmos  *
-- * Funcionais.                                    *
-- *                                               *
-- * ============================================= *

:{

soma :: (Num a) => a -> a -> a
soma valor1 valor2 = valor1 + valor2

x_pi :: (Floating a) => a
x_pi = 3.14159

x_e :: (Floating a) => a
x_e = 2.71828

quadrado :: (Num a) => a -> a
quadrado x = x ^ 2

soma2 :: (Num a) => a -> a -> a
soma2 x y = x + y

quadsoma :: (Num a) => a -> a -> a
quadsoma x y = quadrado (soma2 x y)

peso :: (Floating a) => a
peso = 99999999.49

lognat :: (Floating a) => a -> a
lognat x = peso * (x ** (1 / peso) - 1)

logbas :: (Floating a) => a -> a -> a
logbas x b = lognat (x) / lognat (b)

adicao :: (Eq a, Num a) => a -> a -> a
adicao 0 y = y
adicao x 0 = x
adicao x y = x + y
```

```haskell
par :: (Integral a) => a -> Bool
par n = if mod n 2 == 0 then True else False

impar :: (Integral a) => a -> Bool
impar n = if mod n 2 /= 0 then True else False

impar2 :: (Integral a) => a -> Bool
impar2 n = not (par n)

max :: (Ord a, Num a) => a -> a -> a
max x y = if x > y then x else y

min :: (Ord a, Num a) => a -> a -> a
min x y = if x < y then x else y

negativo :: (Eq a, Ord a, Num a) => a-> a
negativo n = if n < 0 then n else 0 - n

valorx:: (Eq a, Ord a, RealFrac a, Num a) => a -> a
valorx 0 = 0
valorx 1 = 2
valorx n = if n > 1 && n < 9 then n * 5 else n / 5

potencia :: (Eq a, Ord a, Num a) => a -> a -> a
potencia x 0 = 1
potencia x 1 = x
potencia x n = x * potencia x (n - 1)

fib :: (Integral a) => a -> a
fib 0 = 0
fib 1 = 1
fib 2 = 1
fib n = fib (n - 1) + fib (n - 2)

fibbase :: (Integral a) => a -> a -> a -> a
fibbase 0 anterior atual = anterior
fibbase 1 anterior atual = atual
fibbase 2 anterior atual = atual + anterior
fibbase n anterior atual = fibbase (n - 1) atual (anterior + atual)

fib2 :: (Integral a) => a -> a
fib2 n = fibbase n 0 1

mdc :: (Integral a) => a -> a -> a
mdc 0 n = n
mdc m n = mdc (mod n m) m

cabeca :: (Num a) => [a] -> a
cabeca (x : xs) = x

cauda :: (Num a) => [a] -> [a]
cauda (x : xs) = xs

ultimo :: (Num a) => [a] -> a
ultimo [x] = x
ultimo (x : xs) = ultimo xs

arranjo :: (Num a) => [a] -> [a]
arranjo [x] = []
arranjo (x : xs) = x : arranjo xs

somar :: (Num a) => [a] -> a
somar [] = 0
somar (x : xs) = x + somar xs

faixa :: (Ord a, Num a) => a -> a -> a -> [a]
faixa i f p = if i > f
            then []
            else i : faixa (i + p) f p

oposto :: (Num a) => [a] -> [a]
oposto [] = []
oposto (x : xs) = oposto xs ++ [x]

complista :: (Num a) => [a] -> (a -> Bool) -> [a]
complista [] qualificador = []
complista (x : conjunto) qualificador =
  if qualificador x
  then x : complista conjunto qualificador
  else complista conjunto qualificador

listamul :: (Num a) => a -> [a] -> [a]
listamul _ [] = [];
listamul n (x : xs) = (n * x) : listamul n xs

listapot :: (Floating a) => a -> [a] -> [a]
listapot _ [] = [];
listapot n (x : xs) = (x ** n) : listapot n xs

multiplo :: Int -> Int -> Bool
multiplo n m =
  if mod n m == 0 then True else False

divisor :: Int -> [Int]
divisor n = complista (faixa 1 n 1) (multiplo n)

tamanho :: (Num a) => [a] -> Int
tamanho [] = 0
tamanho (x : xs) = 1 + tamanho xs

checa_primo :: Int -> Bool
checa_primo 1 = False
checa_primo 2 = True
checa_primo n =
  if (tamanho (complista (faixa 2 (n - 1) 1) (multiplo n)) > 0)
  then False
  else True

lprimos :: Int -> [Int]
lprimos n = complista (faixa 1 n 1) (checa_primo)

juncao :: (Ord a, Num a) => [a] -> [a] -> [a]
juncao a [] = a
juncao [] b = b
juncao a b = if (cabeca a) < (cabeca b)
            then (cabeca a) : juncao (cauda a) b
            else (cabeca b) : juncao a (cauda b)

possui :: (Eq a, Num a) => [a] -> a -> Bool
possui [] _ = False
possui (x : xs) n = if x == n
                    then True
                    else possui xs n

unico :: (Eq a, Num a) => [a] -> [a]
unico [] = []
unico (x : xs) = if possui xs x
                 then unico xs
                 else x : unico xs

insira :: (Ord a, Num a) => a -> [a] -> [a]
insira n [] = [n]
insira n (x : xs) = if n <= x
                    then n : x : xs
                    else x : insira n xs

classifica :: (Ord a, Num a) => [a] -> [a]
classifica [] = []
classifica (x : xs) = insira x (classifica xs)

uniao :: (Eq a, Ord a, Num a) => [a] -> [a] -> [a]
uniao a b = classifica (unico (juncao a b))

membro :: (Eq a, Num a) => a -> [a] -> Bool
membro _ [] = False
membro a (x : xs) = if a == x then True else membro a xs

interceccao :: (Eq a, Num a) => [a] -> [a] -> [a]
interceccao a [] = []
interceccao [] b = []
interceccao a (x : b) = if membro x a
                        then x : interceccao a b
                        else interceccao a b

diferenca :: (Eq a, Num a) => [a] -> [a] -> [a]
diferenca a [] = a
diferenca [] b = []
diferenca (a : x) b = if membro a b
                      then diferenca x b
                      else a : diferenca x b

igualdade :: (Eq a, Ord a, Num a) => [a] -> [a] -> Bool
igualdade a [] = False
igualdade [] b = False
igualdade (a : as) (b : bs) =
  if classifica (a : as) == classifica (b : bs)
  then True
  else False

sub_lista :: (Eq a, Num a) => [a] -> [a] -> Bool
sub_lista [] [] = True
sub_lista [] _  = True
sub_lista _ [] = False
sub_lista (x : xs) (y : ys) = if x == y
                              then sub_lista xs ys
                              else sub_lista (x : xs) ys

pega_pos :: (Eq a, Num a) => a -> [a] -> Int
pega_pos _ [] = error "elemento nao existe na lista"
pega_pos n (x : xs) = if n == x then tamanho xs else pega_pos n xs

busca :: (Eq a, Num a) => a -> [a] -> Int
busca _ [] = error "lista invalida"
busca n (x : xs) = pega_pos n (oposto (x: xs))

mostra:: (Eq a, Num a) => Int -> [a] -> a
mostra n [] = if n < tamanho [] || n > tamanho []
              then error "indice fora da faixa"
              else 0
mostra 0 (n : xs) = n
mostra n (x : xs) = mostra (n - 1) xs

lista_max :: (Ord a, Num a) => [a] -> a
lista_max [] = error "lista vazia"
lista_max [a] = a
lista_max (x : xs) = if x > lista_max xs
                     then x
                     else lista_max xs

lista_min :: (Ord a, Num a) => [a] -> a
lista_min [] = error "lista vazia"
lista_min [a] = a
lista_min (x : xs) = if x < lista_min xs
                     then x
                     else lista_min xs

replicar :: (Eq a, Num a) => a -> a -> [a]
replicar quantidade valor =
  if quantidade == 0
  then []
  else valor : replicar (quantidade - 1) valor

comeco :: (Ord a, Num a) => Int -> [a] -> [a]
comeco _ [] = []
comeco n (x : xs) = if n > 0
                    then x : comeco (n - 1) xs
                    else []

final :: (Ord a, Num a) => Int -> [a] -> [a]
final _ [] = []
final n (x : xs) = if n - 1 > 0
                   then final (n - 1) xs
                   else xs

separar :: (Ord a, Num a) => [a] -> ([a], [a])
separar [] = ([], [])
separar xs =
  if mod (tamanho xs) 2 /= 0
  then (comeco (div (tamanho xs) 2 + 1) xs,
    final (div (tamanho xs) 2 + 1) xs)
  else (comeco (div (tamanho xs) 2) xs, final (div (tamanho xs) 2) xs)

fatiar :: (Ord a, Num a) => Int -> Int -> [a] -> [a]
fatiar i f x = final i (comeco f x)

mapa :: (Num a) => [a] -> (a -> a) -> [a]
mapa [] funcao = []
mapa (x : xs) funcao = (funcao x) : (mapa xs funcao)

filtro :: (Ord a, Num a) => (a -> Bool) -> [a] -> [a]
filtro funcao [] = []
filtro funcao (x : xs) = if funcao x
                         then x : (filtro funcao xs)
                         else filtro funcao xs

reducao :: (Num a) => [a] ->  (a -> a -> a) -> a -> a
reducao [] funcao n = n
reducao (x : xs) funcao n = funcao x (reducao xs funcao n)

dobra_d :: (Num a) => (a -> a -> a) -> a -> [a] -> a
dobra_d f n [] = n
dobra_d f n (x : xs) = f x (dobra_d f n xs)

dobra_e :: (Num a) => (a -> a -> a) -> a -> [a] -> a
dobra_e f n [] = n
dobra_e f n (x : xs) = dobra_e f (f n x) xs

compacta :: (Num a) => [a] -> [a] -> [(a, a)]
compacta [] b = []
compacta a [] = []
compacta (a : as) (b : bs) = (a, b) : compacta as bs

pares :: (Num a) => [a] -> [(a, a)]
pares xs = compacta xs (cauda xs)

rotac_e :: (Num a) => [a] -> [a]
rotac_e [] = []
rotac_e (x : xs) = xs ++ [x]

rotac_d :: (Num a) => [a] -> [a]
rotac_d [] = []
rotac_d xs = ultimo xs : arranjo xs

-- *** Funcoes bonus ***

dobra_d1 :: (Num a) => (a -> a -> a) -> [a] -> a
dobra_d1 _ [x] = x
dobra_d1 f (x : xs) = f x (dobra_d1 f xs)

dobra_e1 :: (Num a) => (a -> a -> a) -> [a] -> a
dobra_e1  f (x : xs) = dobra_e f x xs

qsort :: (Ord a, Num a) => [a] -> [a] -- quick sort
qsort [] = []
qsort (a : b) = qsort (complista b (<= a))
              ++ [a] ++
              qsort (complista b (> a))

isort :: (Num a, Ord a) => [a] -> [a] -- insertion sort
isort [] = []
isort (x : xs) = insira x (isort xs)

:}

! * ================================================ *
! *                                                  *
! *                Arquivo BONUS                     *
! *   Codigos exemplos: exercicios de aprendizagem   *
! *   ------------------------------------------     *
! *                                                  *
! * Parte integrante do livro: Algoritmos Funcionais *
! * Copyright: Jose Augusto N. G. Manzano            *
! *                                                  *
! * A permissao de uso deste material esta vinculado *
! * a aquisicao do livro correspondente.             *
! *                                                  *
! * Scripts dos codigos para a linguagem Hope        *
! * usados como ilustracao para a fixacao dos con-   *
! * ceitos logicos apresentados no livro Algoritmos  *
! * Funcionais.                                      *
! *                                                  *
! * ================================================ *

dec soma : num # num -> num;
--- soma (valor1, valor2) <= valor1 + valor2;

dec x_pi : num;
--- x_pi <= 3.14159;

dec x_e : num;
--- x_e <= 2.71828;

dec quadrado : num -> num;
--- quadrado x <= pow (x, 2);

dec soma2 : num # num -> num;
--- soma2 (x, y) <= x + y;

dec quadsoma : num # num -> num;
--- quadsoma (x, y) <= quadrado (soma2 (x, y));

dec peso : num;
--- peso <= 99999999.49;

dec lognat : num -> num;
--- lognat x <= peso * (pow (x, 1 / peso) - 1);

dec logbas : num # num -> num;
--- logbas (x, b) <= lognat (x) / lognat (b);

dec adicao : num # num -> num;
--- adicao (0, y) <= y;
--- adicao (x, 0) <= x;
--- adicao (x, y) <= x + y;

dec par : num -> truval;
--- par n <= if n mod 2 = 0 then true else false;

dec impar : num -> truval;
--- impar n <= if n mod 2 /= 0 then true else false;

dec impar2 : num -> truval;
--- impar2 n <= not (par n);

dec max : num # num -> num;
--- max (x, y) <= if x > y then x else y;

dec min : num # num -> num;
--- min (x, y) <= if x < y then x else y;

dec negativo : num -> num;
--- negativo n <= if n < 0 then n else 0 - n;
```

```
dec valorx : num -> num;
--- valorx 0 <= 0;
--- valorx 1 <= 2;
--- valorx n <= if n > 1 and n < 9 then n * 5 else n / 5;

dec potencia : num # num -> num;
--- potencia (x, 0) <= 1;
--- potencia (x, 1) <= x;
--- potencia (x, n) <= x * potencia (x, n - 1);

dec fib : num -> num;
--- fib 0 <= 0;
--- fib 1 <= 1;
--- fib 2 <= 1;
--- fib n <= fib (n - 1) + fib (n - 2);

dec fibbase : num # num # num -> num;
--- fibbase (0, anterior, atual) <= anterior;
--- fibbase (1, anterior, atual) <= atual;
--- fibbase (2, anterior, atual) <= atual + anterior;
--- fibbase (n, anterior, atual) <= fibbase (n - 1, atual, anterior + atual);

dec fib2 : num -> num;
--- fib2 n <= fibbase (n, 0, 1);

dec mdc : num # num -> num;
--- mdc (0, n) <= n;
--- mdc (m, n) <= mdc (n mod m, m);

dec cabeca : list num -> num;
--- cabeca (x :: xs) <= x;

dec cauda : list num -> list num;
--- cauda (x :: xs) <= xs;

dec ultimo : list num -> num;
--- ultimo [x] <= x;
--- ultimo (x :: xs) <= ultimo xs;

dec arranjo : list num -> list num;
--- arranjo [x] <= [];
--- arranjo (x :: xs) <= x :: arranjo xs;

dec somar : list num -> num;
--- somar [] <= 0;
--- somar (x :: xs) <= x + somar xs;

dec faixa : num # num # num -> list num;
--- faixa (i, f, p) <= if i > f
                 then []
                 else i :: faixa (i + p, f, p);
```

```
dec oposto : list num -> list num;
--- oposto [] <= [];
--- oposto (x :: xs) <= oposto xs <> [x];

dec complista : list num # (num -> truval) -> list num;
--- complista ([], qualificador) <= [];
--- complista (x :: conjunto, qualificador) <=
    if qualificador x
    then x :: complista (conjunto, qualificador)
    else complista (conjunto, qualificador);

dec listamul : num # list num -> list num;
--- listamul (_, []) <= [];
--- listamul (n, x :: xs) <= n * x :: listamul (n, xs);

dec listapot: num # list num -> list num;
--- listapot (_, []) <= [];
--- listapot (n, x :: xs) <= pow (x, n) :: listapot (n, xs);

dec multiplo : num # num -> truval;
--- multiplo (n, m) <=
    if n mod m = 0 then true else false;

dec divisor : num -> list num;
--- divisor n <= complista (faixa (1, n, 1), \ d => multiplo (n, d));

dec tamanho : list num -> num;
--- tamanho [] <= 0;
--- tamanho (x :: xs) <= 1 + tamanho (xs);

dec checa_primo : num -> truval;
--- checa_primo 1 <= false;
--- checa_primo 2 <= true;
--- checa_primo n <=
    if tamanho (complista (faixa (2, n - 1, 1), \ d => multiplo (n, d))) > 0
    then false
    else true;

dec lprimos : num -> list num;
--- lprimos n <= complista (faixa (1, n, 1), \ x => checa_primo (x));

dec juncao : list num # list num -> list num;
--- juncao (a, []) <= a;
--- juncao ([], b) <= b;
--- juncao (a, b) <= if cabeca (a) < cabeca (b)
              then cabeca (a) :: juncao (cauda (a), b)
              else (cabeca b) :: juncao (a, cauda (b));

dec possui : list num # num -> truval;
--- possui ([], _) <= false;
--- possui (x :: xs, n) <= if x = n
                  then true
```

```
                 else possui (xs, n);

dec unico : list num -> list num;
--- unico [] <= [];
--- unico (x :: xs) <= if possui (xs, x)
               then unico (xs)
               else x :: unico (xs);

dec insira : num # list num -> list num;
--- insira (n, []) <= [n];
--- insira (n, x :: xs) <= if n =< x
                 then n :: x :: xs
                 else x :: insira (n, xs);

dec classifica : list num -> list num;
--- classifica [] <= [];
--- classifica (x :: xs) <= insira (x, classifica xs);

dec uniao : list num # list num -> list num;
--- uniao (a, b) <= classifica (unico (juncao (a, b)));

dec membro: num # list num -> truval;
--- membro (_, []) <= false;
--- membro (a, x :: xs) <= if a = x then true else membro (a, xs);

dec interceccao : list num # list num -> list num;
--- interceccao (a, []) <= [];
--- interceccao ([], b) <= [];
--- interceccao (a, x :: b) <= if membro (x, a)
                    then x :: interceccao (a, b)
                    else interceccao (a, b);

dec diferenca : list num # list num -> list num;
--- diferenca (a, []) <= a;
--- diferenca ([], b) <= [];
--- diferenca (a :: x, b) <= if membro (a, b)
                   then diferenca (x, b)
                   else a :: diferenca (x, b);

dec igualdade : list num # list num -> truval;
--- igualdade (a, []) <= false;
--- igualdade ([], b) <= false;
--- igualdade (a :: as, b :: bs) <=
    if classifica (a :: as) = classifica (b :: bs)
    then true
    else false;

dec sub_lista : list num # list num -> truval;
--- sub_lista ([], []) <= true;
--- sub_lista ([], _) <= true;
--- sub_lista (_, []) <= false;
--- sub_lista (x :: xs, y :: ys) <= if x = y
```

```
                           then sub_lista (xs, ys)
                           else sub_lista (x :: xs, ys);

dec pega_pos : num # list num -> num;
--- pega_pos (_, []) <= error "elemento nao existe na lista";
--- pega_pos (n, x :: xs) <= if (n = x) then tamanho (xs) else pega_pos (n, xs);

dec busca : num # list num -> num;
--- busca (_, []) <= error "lista invalida";
--- busca (n, x :: xs) <= pega_pos (n, oposto (x :: xs));

dec mostra : num # list num -> num;
--- mostra (n, []) <= if n >= tamanho [] or n < tamanho []
                then error "posicao invalida"
                else 0;
--- mostra (0, n :: xs) <= n;
--- mostra (n, x :: xs) <= mostra (n - 1, xs);

dec lista_max : list num -> num;
--- lista_max [] <= error "lista vazia";
--- lista_max ([a]) <= a;
--- lista_max (x :: xs) <= if x > lista_max xs
                  then x
                    else lista_max xs;

dec lista_min : list num -> num;
--- lista_min [] <= error "lista vazia";
--- lista_min ([a]) <= a;
--- lista_min (x :: xs) <= if x < lista_min xs
                  then x
                    else lista_min xs;

dec replicar : num # num -> list num;
--- replicar (quantidade, valor) <=
    if quantidade = 0
    then []
    else valor :: replicar (quantidade - 1, valor);

dec comeco : num # list num -> list num;
--- comeco (_, []) <= [];
--- comeco (n, x :: xs) <= if n > 0
                    then x :: comeco (n - 1, xs)
                    else [];

dec final : num # list num -> list num;
--- final (_, []) <= [];
--- final (n, x :: xs) <= if n - 1 > 0
                   then final (n - 1, xs)
                   else xs;

dec separar : list num -> list num # list num;
--- separar [] <= ([], []);
--- separar xs <=
    if tamanho (xs) mod 2 /= 0
    then (comeco ((tamanho (xs) div 2) + 1, xs),
      final ((tamanho (xs) div 2) + 1, xs))
    else (comeco (tamanho (xs) div 2, xs), final (tamanho (xs) div 2, xs));

dec fatiar : num # num # list num -> list num;
--- fatiar (i, f, x) <= final (i, comeco (f, x));

dec mapa : list num # (num -> num) -> list num;
--- mapa ([], funcao) <= [];
--- mapa (x :: xs, funcao) <= funcao x :: mapa (xs, funcao);

dec filtro : (num -> truval) # list num -> list num;
--- filtro (funcao, []) <= [];
--- filtro (funcao, x :: xs) <= if funcao x
                      then x :: filtro (funcao, xs)
                      else filtro (funcao, xs);

dec reducao : list num # (num # num -> num) # num -> num;
--- reducao ([], funcao, n) <= n;
--- reducao (x :: xs, funcao, n) <= funcao (x, reducao (xs, funcao, n));

dec dobra_d : (num # num -> num) # num # list num -> num;
--- dobra_d (f, n, []) <= n;
--- dobra_d (f, n, x :: xs) <= f (x, dobra_d (f, n, xs));

dec dobra_e : (num # num -> num) # num # list num -> num;
--- dobra_e (f, n, []) <= n;
--- dobra_e (f, n, x :: xs) <= dobra_e (f, (f (n, x), xs));

dec compacta : list num # list num -> list (num # num);
--- compacta ([], b) <= [];
--- compacta (a, []) <= [];
--- compacta (x :: a, y :: b) <= (x, y) :: compacta (a,  b);

dec pares : list num -> list (num # num);
--- pares xs <= compacta (xs, cauda (xs));

dec rotac_e : list num -> list num;
--- rotac_e [] <= [];
--- rotac_e (x :: xs) <= xs <> [x];

dec rotac_d : list num -> list num;
--- rotac_d [] <= [];
--- rotac_d xs <= ultimo (xs) :: arranjo (xs);

! *** Funcoes bonus ***

dec dobra_d1 : (num # num -> num) # list num -> num;
--- dobra_d1 (_, [x]) <= x;
--- dobra_d1 (f, x :: xs) <= f (x, dobra_d1 (f, xs));
```

```
dec dobra_e1 : (num # num -> num) # list num -> num;
--- dobra_e1  (f, x :: xs) <= dobra_e (f, x, xs);

dec qsort : list num -> list num; !!! quick sort
--- qsort [] <= [];
--- qsort (a :: b) <= qsort (complista (b, \ b => b =< a))
                   <> [a] <>
                   qsort (complista (b, \ b => b > a));

dec isort : list num -> list num; !!! insertion sort
--- isort [] <= [];
--- isort (x :: xs) <= insira (x, isort xs);
```