Problema 3: Máquina de vendas automáticas

Uma máquina de vendas automáticas necessita de dar trocos em moedas conforme a quantia paga e o preço do item comprado.

Pretende-se que programe funções em Haskell para calcular os trocos das transações de máquinas deste tipo.

Exercício 1

Vamos começar por definir uma função para decompor uma quantia em moedas supondo que temos uma quantidade arbitrária de moedas disponíveis.

- Exercício 1: Decompor em moedas

Exercício 2

Vamos agora tornar o problema mais realista assumindo que temos uma lista das moedas disponíveis para trocos antes e depois da transação.

- Exercício 2: Decompor em moedas com estado

Exercício 3

Por fim vamos usar a função do exercício anterior para processar uma lista de vendas na máquina.

- Exercício 3: Processar uma lista de compras

Pedro Vasconcelos, 2022

Exercício 1: Decompor em moedas (simplificado)

Queremos escrever uma função que decompõe uma quantia positiva de euros em moedas. Na máquina de vendas existem moedas de 2€, 1€, 0.50€, 0.20€, 0.10€ e 0.05€. Por exemplo: para decompor 3.45€ devemos dar moedas de 2€, 1€, 0.20€ (duas vezes) e 0.05€.

- Neste exercício assumimos que temos disponíveis quantidades arbitrárias de cada moeda.
- Para evitar erros de arredondamento, vamos representar os preços e valores de moedas em cêntimos (inteiros) em vez de números fracionários de euros.
- A máquina pode dar troco a menos se não possível decompor o valor com as moedas acima.

Escreva uma função

```
decompor :: Int -> [Int]
```

que decompõe uma quantia não-negativa na lista dos valores das moedas ordenados de forma decrescente, eventualmente com repetidos.

Para o exemplo dado, temos que 3.45€ é 345 cêntimos, logo `decompor 345 = [200,100,20,20,5]`. Outros exemplos: `decompor 500 = [200,200,100]`; `decompor 47 = [20,20,5]`.

Sugestão: Comece por definir uma lista (constante) com os valores das moedas disponíveis por ordem decrescente:

```
moedas = [200, 100, 50, 20, 10, 5]
```

Tente exprimir a solução do problema por recursão sobre a quantia que quer decompor, reduzindo-a em cada passo pelo o valor da maior moeda admissível até à quantidade a dar ser inferior a 5 cêntimos.

Exercício 2: Decompor em moedas (com transição de estado)

Vamos agora fazer uma versão da função de decomposição que tem em conta as moedas temos disponíveis antes e depois de fazer trocos (uma *transição de estado*).

Escreva uma função

```
decomporTrans :: Int -> [Int] -> ([Int], [Int])
```

O primeiro argumento é uma quantia não-negativa que queremos decompor; o segundo argumento é a lista com as moedas que estão disponíveis, ordenadas por ordem decrescente (incluido repetidos se temos várias moedas do mesmo valor); o resultado é um par com duas listas:

- a lista de moedas a dar para perfazer a quantia original (se houver moedas suficientes);
- a lista de moedas que sobram.

Por exemplo: `decomporTrans 45 [100,20,20,5,5]` dá `([20,20,5], [100,5])`, ou seja, decompomos 45 em 20+20+5 e sobram dentro da máquina uma moeda de 100 e outra de 5.

Se não estiverem disponíveis moedas suficientes para fazer o troco, devemos dar apenas as que estão disponíveis, sem ultrapassar a quantia original (a máquina nunca dá mais troco do que é devido). Além disso, a máquina deve usar sempre a maior moedas disponível em cada momento, mesmo que isso obrique a fiquar aquém do troco certo.

Um exemplo que exemplifica esta situação: `decomporTrans 60 [50,20,20,20]` dá `([50], [20,20,20])`, ou seja, damos 50 de troco ficando a faltar 10 e sobram dentro da máquina 3 moedas de 20.

Sugestão

Poderá ser útil usar a função `delete :: Eq a => a -> [a] -> [a]` do módulo `Data.List` para eliminar cada moeda escolhida da lista de moedas disponíveis.

Exemplo:

```
delete 3 [1,2,3,3,4] == [1,2,3,4]
```

Exercício 3: Processar transações

Vamos agora escrever uma função que simula o processamento de uma transação da máquina de vendas.

- A transação é representada por um par `(Int,[Int])` de um preço a pagar e a lista das moedas introduzidas
- Se a soma dos valores das moedas introduzidas for igual ou superior ao preço então a máquina deve aceitar as moedas e devolver troco
- Se a soma dos valor das moedas introduzidas for inferior ao preço, então a máquina deve devolver as moedas introduzidas

Pretende-se que escreva uma função

```
transação :: (Int,[Int]) -> [Int] -> ([Int], [Int])
```

O primeiro argumento é uma transação a efetuar; o segundo argumento é a lista inicial das moedas disponíveis na máquina; o resultado é um par com a lista de moedas devolvidas e a lista final de moedas na máquina. Deve manter o invariante de que as listas de moedas estão sempre ordenadas por ordem decrescente (eventualmente com repetidos).

Ao dar trocos deve sempre usar as maiores moedas disponíveis. Note que isto pode fazer com que a máquina dê troco a menos mesmo quando tem as moedas necessárias para dar troco certo (ver Exemplo 2).

Exemplo 1

```
transação (120, [100,50]) [50,20,10,5] == ([20,10], [100,50,50,5])
```

Explicação:

- a máquina começa com moedas de 50, 20, 10, 5;
- recebe 150 para pagar 120; tem moedas para fazer troco certo (30)
- no final sobram na máquina moedas de 100, 50, 50, 5

Exemplo 2

```
transação (40, [100]) [50,20,20,20] = ([50], [100,20,20,20])
```

Exemplo 3

```
transação (120, [100,10]) [50,20,10] = ([100,10], [50,20,10])
```

Sugestões

- Pode fazer o troco usando a função `decomporTrans` do exercício anterior; inclua-a na sua submissão e use-a como função auxiliar
- Outras funções auxiliares do módulo `Data.List` que podem ser úteis: `sort` (para ordenar uma lista ordem crescente) e `reverse` (para colocar por ordem inversa).