Programmation fonctionnelle (L3)
TP noté (2h00)

Stéphane Vialette `stephane.vialette@univ-eiffel.fr`

20 novembre 2025

Les deux exercices sont indépendants. Seuls les documents du cours et une feuille recto-verso manuscrite sont autorisés.

Dans la mesure du possible, ne pas écrire du code grossièrement inefficace (pas de `[x] ++ xs` lorsque `x : xs` fait sens par exemple !). Au sein d'un même exercice, l'ordre des questions n'est jamais vraiment anodin ; il peut être utile de se référer à une ou des questions précédantes pour écrire une nouvelle fonction.

Votre TP doit impérativement se trouver dans le répertoire `EXAM` sous la forme d'un unique fichier nommé `TPNote.hs`. Un squelette de même nom vous est fourni.

Exercice 1 : Matrices

Une *matrice*, c'est simplement un tableau rectangulaire de nombres, organisé en *lignes* (horizontalement) et en *colonnes* (verticalement). Par exemple :

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

Ici, la matrice a 3 lignes et 4 colonnes. Chaque nombre de la matrice est appelé un *élément*, et on peut le repérer selon sa position dans la matrice (sa ligne et sa colonne). Un *vecteur ligne* est une matrice qui ne contient qu'une seule ligne. Un *vecteur colonne*, c'est l'inverse : il ne contient qu'une seule colonne. Une *matrice carrée* est une matrice qui a le même nombre de lignes et de colonnes. Une matrice carrée est dite *triangulaire inférieure* si tous les éléments au-dessus de la diagonale principale sont nuls. Par exemple :

$$B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 10 \end{pmatrix}$$

est une matrice triangulaire inférieure.

Les matrices sont très utiles pour représenter des données, résoudre des systèmes d'équations, ou effectuer des calculs en informatique, en physique, et bien sûr ... en Haskell !

Dans la suite, on représente une matrice par une liste de listes de même longueur, où chaque liste interne représente une ligne de la matrice. Par exemple, la matrice A ci-dessus est représentée par la liste de listes : `[[1,2,3,4],[5,6,7,8],[9,10,11,12]]`. La matrice B est représentée par la liste de listes : `[[1,0,0,0],[2,3,0,0],[4,5,6,0],[7,8,9,10]]`. On suppose que les matrices sont toujours bien formées, c'est-à-dire que dans la représentation par une liste de listes toutes les listes internes ont la même longueur.

Dans la suite, afin d'illustrer les différentes fonctions demandées, on considère les matrices suivantes :

```haskell
m1 :: [[Int]]
m1 = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]

m2 :: [[Int]]
m2 = [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]

m3 :: [[Int]]
m3 = [[1,2,3,4,5,6]]

m4 :: [[Int]]
m4 = [[1],[2],[3],[4],[5],[6]]

ms :: [[[Int]]]
ms = [m1, m2, m3, m4]
```

(a) Écrire la fonction :

```haskell
dim :: [[a]] -> (Int, Int)
```

qui calcule la *dimension* d'une matrice donnée. La dimension d'une matrice A est un couple `(m, n)` où `m` est le nombre de lignes de A et `n` le nombre de colonnes de A.

```
ghci> [dim m | m <- ms]
[(3,4),(4,4),(1,6),(6,1)]
```

(b) Écrire les prédicats :

```haskell
square :: [[a]] -> Bool
rVect :: [[a]] -> Bool
cVect :: [[a]] -> Bool
```

qui testent respectivement si une matrice est carrée, si elle est un vecteur ligne, ou si elle est un vecteur colonne.

```
ghci> [square m | m <- ms]
[False,True,False,False]
ghci> [rVect m | m <- ms]
[False,False,True,False]
ghci> [cVect m | m <- ms]
[False,False,False,True]
```

(c) Écrire le prédicat :

```haskell
conformableWith :: [[a]] -> [[a]] -> Bool
```

qui teste si deux matrices sont *conformables* pour la multiplication. Deux matrices sont conformables pour la multiplication si le nombre de colonnes de la première est égal au nombre de lignes de la seconde.

```
ghci> m1 `conformableWith` m2
True
ghci> m2 `conformableWith` m1
False
ghci> m3 `conformableWith` m4
True
ghci> m4 `conformableWith` m3
True
```

(d) Écrire la fonction :

```haskell
transposeM :: [[a]] -> [[a]]
```

qui calcule la *transposée* d'une matrice donnée. La transposée d'une matrice est obtenue en échangeant ses lignes et ses colonnes.

```
ghci> transposeM m1
[[1,5,9],[2,6,10],[3,7,11],[4,8,12]]
ghci> transposeM m2
[[1,5,9,13],[2,6,10,14],[3,7,11,15],[4,8,12,16]]
ghci> transposeM m3
[[1],[2],[3],[4],[5],[6]]
ghci> transposeM m4
[[1,2,3,4,5,6]]
```

(e) Écrire la fonction :

```
mult :: Num a => [[a]] -> [[a]] -> [[a]]
```

qui calcule le *produit* de deux matrices conformables. Le produit de deux matrices `A` et `B` est une matrice `C` telle que chaque élément `C[i][j]` (*i.e.* c'est une notation informelle pour désigner l'élément de la ligne `i` et de la colonne `j` de `C`) est la somme des produits des éléments de la ligne `i` de `A` par les éléments de la colonne `j` de `B`. On supposera dans l'écriture de la fonction que les deux matrices sont toujours conformables.

```
ghci> mult m1 m2
[[90,100,110,120],[202,228,254,280],[314,356,398,440]]
ghci> mult m3 m4
[[91]]
```

(f) Écrire la fonction :

```
subsets :: Int ->  [a] -> [[a]]
```

qui calcule toutes les sous-listes d'une liste donnée de taille `k`.

```
ghci> subsets 0 "abcd"
[""]
ghci> subsets 2 "abcd"
["ab","ac","ad","bc","bd","cd"]
ghci> subsets 3 "abcde"
["abc","abd","abe","acd","ace","ade","bcd","bce","bde","cde"]
ghci> subsets 5 "abcde"
["abcde"]
```

(g) Écrire la fonction :

```
select :: [Int] -> [a] -> [a]
```

qui sélectionne les éléments d'une liste aux positions indiquées par une liste d'indices. On suppose que les indices sont valides et distincts. Le premier élément de la liste correspond à l'indice `0`.

```
ghci> select [0,2,4] "abcdef"
"ace"
ghci> select [1,3,5] "abcdef"
"bdf"
```

(h) Écrire la fonction :

```
subs :: Int -> Int -> [[a]] -> [[[a]]]
```

qui calcule toutes les *sous-matrices* de taille $\mathtt{m} \times \mathtt{n}$ d'une matrice donnée. Une sous-matrice de taille $\mathtt{m} \times \mathtt{n}$ d'une matrice `A` est obtenue en sélectionnant $\mathtt{m}$ lignes de `A` et $\mathtt{n}$ colonnes de `A`. On suppose que $\mathtt{m}$ et $\mathtt{n}$ sont toujours valides.

```
ghci> subs 2 2 m1
[[[1,2],[5,6]],[[1,2],[9,10]],[[1,3],[5,7]],[[1,3],[9,11]],
 [[1,4],[5,8]],[[1,4],[9,12]],[[2,3],[6,7]],[[2,3],[10,11]],
 [[2,4],[6,8]],[[2,4],[10,12]],[[3,4],[7,8]],[[3,4],[11,12]]]
ghci> subs 2 4 m1
```

```
[[[1,2,3,4],[5,6,7,8]],[[1,2,3,4],[9,10,11,12]],[[5,6,7,8],[9,10,11,12]]]
```

(i) Déduire de la question précédante l'écriture de la fonction :
`sub :: Eq a => [[a]] -> [[a]] -> Bool`
qui teste si une matrice est une sous-matrice d'une autre matrice.

```
ghci> sub [[1,2,4],[9,10,12]] m1
True
ghci> sub [[6,7],[10,11]] m1
True
ghci> sub [[6,7],[9,11]] m1
False
```

(j) Écrire le prédicat :
`isLowerTriangular :: (Eq a, Num a) => [[a]] -> Bool`
qui teste si une matrice carrée est triangulaire inférieure. On admet sans le tester dans la fonction que la matrice est carrée.

```
ghci> isLowerTriangular [[1,0,0],[2,3,0],[4,5,6]]
True
ghci> isLowerTriangular [[1,0,0],[2,0,3],[4,5,6]]
False
ghci> isLowerTriangular m1
False
```

Exercice 2 : Les pancakes déterministes

Le *problème des pancakes* a été introduit par Donald Knuth dans les années 1970 comme un problème d'algorithmique amusant et pédagogique. Dans sa version classique, on dispose d'une pile de pancakes de tailles différentes, et on cherche à les trier du plus grand en bas au plus petit en haut, en utilisant uniquement une opération appelée *flip* : on retourne les premiers k pancakes de la pile à l'aide d'une spatule. Par exemple, si la pile initiale est

$$5, 3, 8, 2, 9, 1, 7, 4, 6$$

(le pancake du haut porte le numéro 5, celui en dessous porte le numéro 3, etc.), et si on effectue un flip avec $k = 4$, on obtient la pile

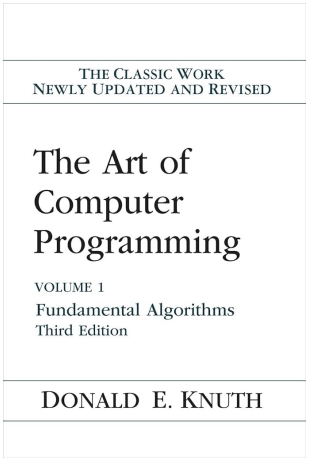
$$2, 8, 3, 5, 9, 1, 7, 4, 6$$

(les 4 premiers pancakes ont été retournés).

Dans la version *déterministe simplifiée* (la version qui nous concerne aujourd'hui), on s'intéresse uniquement à placer le pancake numéro 1 en tête de la pile. De plus, le processus est complètement déterministe :

- On commence avec une liste d'entiers représentant les pancakes empilés les uns sur les autres. Par exemple, la liste `[5,3,8,2,9,1,7,4,6]` représente une pile de pancakes où le pancake du haut porte le numéro 5, le pancake en dessous porte le numéro 3, etc.
- On effectue une série de flips selon la procédure suivante :
  - On considère uniquement le pancake en tête de la pile, qui porte le numéro k.
  - On retourne les k premiers pancakes de la pile.
  - On répète le processus et on s'arrête lorsque le pancake numéro 1 est en tête de la pile [1].

1. Petite remarque de culture générale : même si la terminaison de ce processus est garantie (*i.e.*, on peut prouver que le pancake numéro 1 finit toujours par arriver en première position), on ne sait pas si le nombre d'étapes nécessaires est toujours borné par un polynôme en la taille n de la pile. Autrement dit, on ignore si, dans le pire des cas, cet algorithme pourrait nécessiter un nombre d'opérations qui croît très rapidement (par exemple de manière exponentielle) par rapport à la taille de la permutation.



Par exemple, en partant de la pile initiale

$$5, 3, 8, 2, 9, 1, 7, 4, 6$$

on effectue les flips suivants :

$$9, 2, 8, 3, 5, 1, 7, 4, 6$$
$$6, 4, 7, 1, 5, 3, 8, 2, 9$$
$$3, 5, 1, 7, 4, 6, 8, 2, 9$$
$$1, 5, 3, 7, 4, 6, 8, 2, 9$$

et on s'arrête car le pancake numéro 1 est en tête de la pile.

(a) Écrire la fonction

```
reversePrefix :: Int -> [a] -> [a]
```

qui retourne une nouvelle liste où les k premiers éléments d'une liste donnée sont renversés. Si l'entier k est plus grand que la longueur de la liste, on renverse toute la liste. Si l'entier k est négatif ou nul, on retourne la liste inchangée.

```
ghci> reversePrefix 0 [5,3,8,2,9,1,7,4,6]     -- liste inchangee
[5,3,8,2,9,1,7,4,6]
ghci> reversePrefix (-3) [5,3,8,2,9,1,7,4,6]  -- liste inchangee
[5,3,8,2,9,1,7,4,6]
ghci> reversePrefix 2 [5,3,8,2,9,1,7,4,6]
[3,5,8,2,9,1,7,4,6]
ghci> reversePrefix 4 [5,3,8,2,9,1,7,4,6]
[2,8,3,5,9,1,7,4,6]
ghci> reversePrefix 9 [5,3,8,2,9,1,7,4,6]
[6,4,7,1,9,2,8,3,5]
ghci> reversePrefix 10 [5,3,8,2,9,1,7,4,6]
[6,4,7,1,9,2,8,3,5]
```

(b) Écrire le prédicat

```
final :: (Eq a, Num a) => [a] -> Bool
```

qui retourne vrai si et seulement si le premier élément de la liste est égal à 1.

```
ghci> final []
False
ghci> final [5,3,8,2,9,1,7,4,6]
False
```

```
ghci> final [1,5,3,7,4,6,8,2,9]
True
```

(c) Écrire la fonction

```
deterministicPancakeFlipping :: [Int] -> [[Int]]
```

qui retourne la liste des étapes successives du processus de flips déterministes, en commençant par la pile initiale. La liste retournée doit contenir la pile initiale, puis chaque pile obtenue après chaque flip, et se terminer par la pile finale où le pancake numéro 1 est en tête de la pile. Cet ordre est imposé (il doit donc être respecté).

```
ghci> deterministicPancakeFlipping [5,3,8,2,9,1,7,4,6]
[[5,3,8,2,9,1,7,4,6],
 [9,2,8,3,5,1,7,4,6],
 [6,4,7,1,5,3,8,2,9],
 [3,5,1,7,4,6,8,2,9],
 [1,5,3,7,4,6,8,2,9]]
ghci> deterministicPancakeFlipping [1,4,3,2,8,6,5,7,9]
[[1,4,3,2,8,6,5,7,9]]
```

(d) Écrire la fonction

```
maxBy :: Ord b => (a -> b) -> [a] -> [a]
```

qui, étant donné une fonction de coût `f` et une liste d'éléments, retourne la liste de tous les éléments maximaux relativement à `f`.

```
ghci> maxBy length ["a","ab","abc","de","ij","klmno","fgh"]
["klmno"]
ghci> maxBy sum [[1,2,3],[4,5],[6],[7,8,9],[10]]
[[7,8,9]]
ghci> maxBy sum [[1,2,3],[10],[4,5],[7,3],[6]]
[[10],[7,3]]
ghci> maxBy (`mod` 3) [1..9]
[2,5,8]
```

(e) Pour une taille n de pile donnée, on recherche toutes les piles initiales de taille n qui maximisent le nombre de flips nécessaires pour amener le pancake numéro 1 en tête de la pile. Écrire la fonction :

```
maxDeterministicPancakeFlippings :: Int -> [[[Int]]]
```

qui retourne la liste de toutes les piles initiales de taille n qui nécessitent le plus grand nombre de flips pour amener le pancake numéro 1 en tête de la pile. On rappelle que la fonction `Data.List.permutations :: [a] -> [[a]]` peut être utilisée pour générer toutes les permutations d'une liste [2].

```
ghci> maxDeterministicPancakeFlippings 3
[[2,3,1],[3,1,2]]
ghci> maxDeterministicPancakeFlippings 4
[[2,4,1,3],[3,1,4,2]]
ghci> maxDeterministicPancakeFlippings 5
[[3,1,4,5,2]]
ghci> maxDeterministicPancakeFlippings 6
[[5,6,4,1,3,2],[3,6,5,1,4,2],[4,1,6,5,2,3],[4,1,5,2,6,3],[4,5,6,2,1,3]]
ghci> maxDeterministicPancakeFlippings 7
[[3,1,4,6,7,5,2],[4,7,6,2,1,5,3]]
```

2. Si la fonction `Data.List.permutations` n'est pas importée dans `Prelude`, il faut importer le module `Data.List` par un simple `import Data.List` ou `import qualified Data.List as L` pour y avoir accès.

(f) On dit qu'une pile A est un *ancêtre* d'une pile B si la pile B apparaît dans la liste des étapes successives du processus de flips déterministes à partir de la pile A. Par exemple, la pile

$$5, 3, 8, 2, 9, 1, 7, 4, 6$$

est un ancêtre de la pile

$$3, 5, 1, 7, 4, 6, 8, 2, 9$$

car cette dernière apparaît dans la liste des étapes successives du processus de flips déterministes à partir de la première pile.

Écrire le prédicat :

```
ancestorOf :: [Int] -> [Int] -> Bool
```

qui teste si une pile est un ancêtre d'une autre pile.

```
ghci> [5,3,8,2,9,1,7,4,6] `ancestorOf` [3,5,1,7,4,6,8,2,9]
True
ghci> [5,3,8,2,9,1,7,4,6] `ancestorOf` [1,2,3,4,5,6,7,8,9]
False
```

(g) Deux piles sont dites *jumelles* si elles produisent, après une même nombre de flips déterministes, une même pile. Par exemple, les piles initiales

$$5, 3, 8, 2, 9, 1, 7, 4, 6$$

et

$$6, 4, 7, 1, 9, 2, 8, 3, 5$$

sont jumelles car après 3 flips, elles produisent toutes les deux la pile

$$3, 5, 1, 7, 4, 6, 8, 2, 9.$$

Écrire le prédicat

```
twin :: [Int] -> [Int] -> Bool
```

qui teste si deux piles sont jumelles.

```
ghci> twin [5,3,8,2,9,1,7,4,6] [6,4,7,1,9,2,8,3,5]
True
ghci> twin [5,3,8,2,9,1,7,4,6] [1,2,3,4,5,6,7,8,9]
False
```