Machine Exams

Les consignes de rendu

Créez le **dossier de rendu** : *prenom.nom* (votre login), dans lequel vous devez fournir un fichier exo$i$.ml par exercice demandé contenant chacun les sources **(sans les évaluations de CAML !)** de l'exercice correspondant.

```
firstname.lastname.zip
+-- firstname.lastname/
  |-- exo1.ml
  |-- exo2.ml
  |-- exo3.ml
  |-- exo4.ml
  |-- exo5.ml
```

Certains exercices nécessitent des fonctions du précédent TP : elles doivent obligatoirement être incluses dans le `.ml` de l'exercice concerné.

1 Midterm S1 nov. 2017

Exercice 1 : PGCD par décomposition en facteurs premiers

Le pgcd de deux entiers u et v peut se calculer de la manière suivante :
— en décomposant les nombres u et v en facteurs premiers,
— puis en calculant le produit des facteurs communs.

decompose

On désire construire la liste des facteurs premiers d'un entier naturel n. Une méthode simple consiste à diviser successivement l'entier par tous les entiers d tels que $2 \leq d < n$.

Utiliser cette méthode pour écrire la fonction `decompose n` qui retourne la liste des facteurs premiers de `n` triés en ordre croissant lorsque `n` est supérieur à 2.

```
val decompose : int -> int list = <fun>

# decompose 45 ;;
- : int list = [3; 3; 5]

# decompose 150 ;;
- : int list = [2; 3; 5; 5]

# decompose 0 ;;
Exception: Invalid_argument "decompose: parameter <= 1".
```

shared

Soient deux listes d'éléments croissantes. Écrire une fonction qui construit la liste des éléments communs aux deux listes.

```
val shared : 'a list -> 'a list -> 'a list = <fun>

# shared [1; 3; 6; 6; 8; 9] [2; 5; 6; 6; 8; 9] ;;
- : int list = [6; 6; 8; 9]

# shared [2; 3; 5; 5; 5] [3; 3; 5] ;;
- : int list = [3; 5]

# shared [3; 3; 5; 7] [2; 3; 3; 3; 7; 7] ;;
- : int list = [3; 3; 7]
```

gcd

Écrire la fonction `gcd` qui calcule le pgcd de deux entiers supérieurs à 2 en calculant le produit de leurs facteurs communs.

```
val gcd : int -> int -> int = <fun>

# gcd 45 150;;
- : int = 15

# gcd 100 7;;
- : int = 1
```

Exercice 2 : Immédiatement décodable

Ici, un ensemble de symboles codés est dit *immédiatement décodable* si aucun code des symboles ne sert de préfixe à un autre code du même ensemble.

Par exemple : Si l'on considère un alphabet constitué des symboles {A,B,C,D},
- l'ensemble de codes suivant est *immédiatement décodable* :
  `A:01  B:10  C:0010  D:0000`
- mais celui-ci ne l'est pas :
  `A:01  B:10  C:010  D:0000` (A sert de préfixe à C)

Chaque code sera représenté par une liste d'entiers. Un ensemble de codes sera une liste de codes.

is_prefix

Écrire une fonction `is_prefix` qui prend en paramètre une liste l et une liste de listes ll et qui vérifie si l est préfixe d'une des sous-listes de ll, ou si une des sous-listes de ll est préfixe de l.

```
val is_prefix : 'a list -> 'a list list -> bool = <fun>

# is_prefix [1;2] [[2]; [2;1]; [1;2;3]] ;;
- : bool = true
# is_prefix [1;2] [[3;1]; [1;3]; [2;1]] ;;
- : bool = false
# is_prefix [1;2] [[3;1]; [1]; [2;1]] ;;
- : bool = true
# is_prefix [] [] ;;
- : bool = false
# is_prefix [] [[]] ;;
- : bool = true
```

decodable

Écrire une fonction `decodable` qui prend en paramètre un ensemble de codes (représenté par une liste de codes) et qui détermine si cet ensemble est *immédiatement décodable*.

```
val decodable : 'a list list -> bool = <fun>

# decodable [[0;1]; [1;0]; [0;0;1;0]; [0;0;0;0]] ;;
- : bool = true
# decodable [[0;1]; [1;0]; [0;1;0]; [0;0;0;0]] ;;
- : bool = false
# decodable [[0]; [1;0]; [1;1;0]; [1;1;1]] ;;
- : bool = true
# decodable [[1;0]; [0]; [1;0;0]] ;;
- : bool = false
```

Exercice 3 : Les entiers longs

On peut représenter un entier naturel comme étant une liste de chiffres, commençant par le chiffre des unités, puis celui des dizaines...
Par exemple l'entier `987654321` sera représenté par la liste : `[1; 2; 3; 4; 5; 6; 7; 8; 9]`. Dans la suite cette représentation sera appelée *entier long*.

bigint_sum

Écrire la fonction `bigint_sum` qui additionne deux *entiers longs* (attention aux retenues).

```
val bigint_sum : int list -> int list -> int list = <fun>

# bigint_sum [0; 1; 2; 5] [8; 2; 9] ;;
- : int list = [8; 3; 1; 6]
```

$$\begin{array}{rrrrr} & _{1} & & & \\ & 5 & 2 & 1 & 0 \\ + & & 9 & 2 & 8 \\ \hline & 6 & 1 & 3 & 8 \end{array}$$

bigint_mult

Écrire une fonction `bigint_mult` qui multiplie un *entier long* par un entier naturel (`int`).

```
val bigint_mult : int list -> int -> int list = <fun>

# bigint_mult [1; 2; 3; 4; 5] 5 ;;
- : int list = [5; 0; 6; 1; 7; 2]
```

$$\begin{array}{rrrrrr} & & _{2} & _{1} & _{1} & & \\ & & 5 & 4 & 3 & 2 & 1 \\ \times & & & & & & 5 \\ \hline & 2 & 7 & 1 & 6 & 0 & 5 \end{array}$$

bigint_times

Écrire une fonction `bigint_times` qui multiplie deux *entiers longs*.

```
val bigint_times : int list -> int list -> int list = <fun>

# bigint_times [0; 1; 2; 5] [8; 2; 9] ;;
- : int list = [0; 8; 8; 4; 3; 8; 4]
```

Rappel :

$$5210 \times 928 = 834880$$

$$car \quad \begin{array}{rl} & 8 \times 5210 \\ + & (2 \times 5210) \times 10 \\ + & (9 \times 5210) \times 100 \\ \hline = & 834880 \end{array}$$

2 Midterm S1# avril 2017

Exercice 4 : Notation positionnelle

*"La **notation positionnelle** est un procédé d'écriture des nombres, dans lequel chaque position est reliée à la position voisine par un multiplicateur, appelé base du système de numération. [...] La valeur d'une position est celle du symbole de position ou celle de la précédente position apparente multipliée par la base. Le nombre de symboles nécessaires est au moins égal à la base ou à la plus grande base auxiliaire utilisée. Le système décimal usuel utilise dix symboles (0,1,2,3,4,5,6,7,8,9)"*
Source : Wikipédia.
Exemples :
- Décomposition en base 2 de 6 : `['1'; '1'; '0']`
  ($6 = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$).
- Décomposition en base 10 de 2563 : `['2'; '5'; '6'; '3']`
  ($2563 = 2 \times 10^3 + 5 \times 10^2 + 6 \times 10^1 + 3 \times 10^0$).
- Décomposition en base 16 de 2000 : `['7'; 'D'; '0']`
  ($2000 = 7 \times 16^2 + 13 \times 16^1 + 0 \times 16^0$).

Pour l'exercice nous représenterons la base sous forme d'une liste de caractères. Donc la base sera égale au nombre d'éléments dans la liste.

Seule la position du caractère dans la base est importante.

Exemples de bases :

- Base 2 : `['0';'1']`
- Base 16 : `['0';'1';'2';'3';'4';'5';'6';'7';'8';'9';'A';'B';'C';'D';'E';'F']`
- Base 4 : `['W';'X';'Y';'Z']`

decompose

Écrire la fonction `decompose x b` qui retourne le nombre `x` décomposé en éléments de la base `b`.

```
val decompose : int -> 'a list -> 'a list = <fun>

# decompose 42 ['0';'1';'2';'3';'4';'5';'6';'7';'8';'9'];;
- : char list = ['4'; '2']

# decompose 1337 ['W';'X';'Y';'Z'];;
- : char list = ['X'; 'X'; 'W'; 'Z'; 'Y'; 'X']
```

recompose

Écrire la fonction `recompose l b` qui retourne le nombre recomposé à partir de la liste d'éléments `l` et de la base `b`.

```
val recompose : 'a list -> 'a list -> int = <fun>

# recompose ['7';'2';'0';'2';'7'] ['9';'8';'7';'6';'5';'4';'3';'2';'1';'0'];;
- : int = 27972

# recompose ['4';'2'] ['1';'0'];;
Exception: Failure "out of base".
```

Exercice 5 : Séquence de Lucas

Le résumé de la séquence de Lucas est une suite où chaque terme est contruit en décrivant brièvement les valeurs des deux termes précédents.

rang	résumé
1	1
2	3
3	1 3 1 1
4	2 3 3 1
5	3 3 1 2 4 1
6	1 4 4 3 2 2 3 1

Ainsi par exemple le rang 4 est contruit à partir des rangs 2 (3) et 3 (1 3 1 1). On constate que le chiffre 3 est présent 2 fois, puis que le chiffre 1 est présent 3 fois. On construira donc le résumé pour le rang 4 suivant : `2 3 3 1`.

On commencera toujours par résumer les chiffres les plus grands.

insert

Écrire la fonction `insert` qui insère un élément à sa place dans une liste triée en ordre décroissant.

```
val insert : 'a -> 'a list -> 'a list = <fun>

# insert 2 [8; 6; 4; 1] ;;
- : int list = [8; 6; 4; 2; 1]
```

sort

Écrire la fonction `sort` qui trie une liste, de façon décroissante.

```
val sort : 'a list -> 'a list = <fun>

# sort [1; 3; 4; 6; 7; 0; 2] ;;
- : int list = [7; 6; 4; 3; 2; 1; 0]
```

count

Écrire la fonction `count` qui compte le nombre d'éléments consécutifs identiques, et retourne une liste contenant ce comptage. On n'appelera cette fonction qu'avec une liste triée en ordre décroissant. L'ordre de sortie dans la liste est : nombre d'occurrences puis valeur.

```
val count : int list -> int list = <fun>

# count [1] ;;
- : int list = [1; 1]

# count [4; 4; 4; 3; 3; 1; 1] ;;
- : int list = [3; 4; 2; 3; 2; 1]
```

Lucas

Écrire la fonction `lucas n` qui construit le résumé de la séquence au rang `n`.

```
val lucas : int -> int list = <fun>

# lucas 1 ;;
- : int list = [1]

# lucas 2 ;;
- : int list = [3]

# lucas 5 ;;
- : int list = [3; 3; 1; 2; 4; 1]

# lucas 8 ;;
- : int list = [5; 4; 5; 3; 3; 2; 3; 1]
```