Chapitre 2

Listes

Option informatique MP1, MP2 & MP3

I Définition

I.1 Retour sur les tableaux

Nous avons défini les tableaux dans le chapitre précédent.
Les avantages des tableaux sont nombreux.

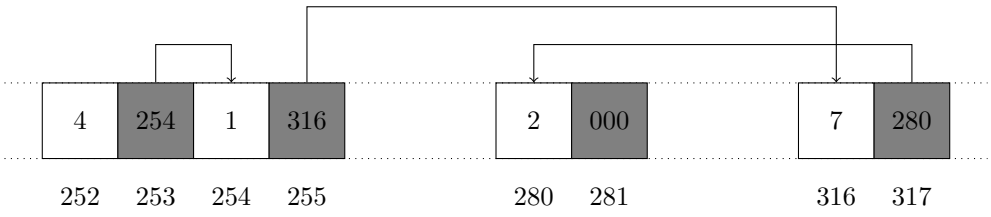
- L'encombrement en mémoire est limité : on ne conserve que la valeur stockée.
- L'accès aux éléments est rapide : la valeur est lue sans intermédiaire. De plus la contiguïté des valeurs permet le déplacement des valeurs suivantes dans les mémoires-cache et accélère la lecture.
- La modification d'une valeur est immédiate, l'ancienne valeur disparaît.

Cependant les tableaux ont aussi des inconvénients.

- Le premier inconvénient est le nombre d'élément qui doit être choisi à l'avance, il n'est pas possible d'ajouter un nouvel emplacement [1].
- La modification d'une valeur élimine la valeur précédente : il n'y a pas persistance des données.

I.2 Définition

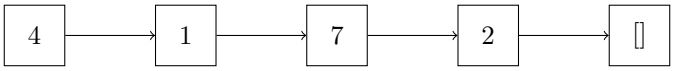
Les listes donnent une solution aux inconvénient indiqués ci-dessus mais sont moins efficaces que les tableaux en termes d'occupation mémoire et de vitesse d'accès.
Une liste est implémentée dans un ordinateur sous la forme d'une liste chaînée : chaque valeur est couplée à une adresse qui indique où est la valeur suivante.
Par exemple une liste formée de 4, 1, 7 et 2 peut être implémentée sous la forme



L'adresse `000` ne correspond pas à une adresse physique ; elle signifie qu'il n'y a plus de terme ensuite.
La liste sera associée à l'adresse 252 ; on y lit la première valeur, 4.
Pour connaître la valeur suivante on doit lire l'adresse que l'on lit en $252+1$, c'est 254. Le troisième terme sera trouvé à l'adresse trouvée en $254+1$ donc en 316 ...

Une représentation abstraite est possible en n'indiquant pas les adresses physiques mais en liant les valeurs.

1. Les tableaux Python permettent la modification de la taille mais cela se fait au prix d'un encombrement en mémoire et d'un temps d'accès augmentés.



C'est cette abstraction qui sera utilisée dans les langages de haut niveau.
On peut remarquer que ce que l'on connait de la liste est la première valeur et le reste de la liste qui est aussi une liste (éventuellement vide).

Définition 1 - Listes Caml

Une α-liste est

- soit la liste vide, notée `[]`,
- soit l'assemblage, noté `t::q`, d'un élément t, de type α, et d'une α-liste q. t est la **tête** de la liste, q en est la **queue**.

La liste ci dessus peut donc être définie par `let liste = 4::1::7::2::[];;`.
Elle sera affichée sous la forme `[4; 1; 7; 2]`.
On pouvait la définir directement sous cette forme `let liste = [4; 1; 7; 2];;`.
Il faut noter que les éléments d'une liste ont un type qui doit être constant : une liste est un assemblage homogène.

II Utilisation

II.1 Déconstruction

Pour voir les termes d'une liste il faut déconstruire l'assemblage.
La tête et la queue d'une liste sont obtenus par les fonctions `List.hd` (pour **head**) et `List.tl` (pour **tail**). Ces fonctions renvoient une erreur si la liste est vide.
Comme la construction d'une liste est récursive le moyen naturel pour les utiliser est une fonction récursive. Le cas terminal sera souvent le cas d'une liste vide.
Par exemple la somme des termes d'une liste d'entiers est calculée par

```
let rec somme liste =
  if liste = []
  then 0
  else List.hd liste + (somme (List.tl liste));;
```

OCaml permet une construction simplifiée du test ci-dessus : le **pattern-matching**.
Les listes ont deux structures possibles, vide ou un assemblage, on parle de **motifs** (patterns en anglais).
On peut filtrer les différents motifs d'une variable par la construction

```
...
  match variable with
   motif1 -> instructions1
  |motif2 -> instructions2
  |motif3 -> instructions3
  ...
```

La barre verticale symbolise un "ou".
Il est possible (et peut être plus lisible) de mettre ce symbole aussi devant le premier motif.
Dans le cas d'une liste les motifs seront souvent `[]` et `t::q`
mais d'autres motifs sont possibles : `a::b::q`, `0::q`, ...

```
let rec somme liste =
  match liste with
  |[] -> 0
  |t::q -> t + (somme q);;
```

- Un motif peut contenir une valeur constante mais pas la valeur d'une variable.
- Quand on veut filtrer en utilisant une variable on peut écrire une condition `if` mais on peut aussi ajouter une condition après le motif avec le mot-clé `when`. On doit reprendre le même motif ensuite pour gérer les cas où la condition n'est pas vérifiée.

```
  match variable with
  |motif1 -> instructions1
  |motif2 when condition -> instructions2
  |motif2 -> instructions3
  ...
```

- Les filtres sont appliqués du haut vers le bas, la fonction est appliquée au premier filtre satisfait.
- Les cas restant peuvent être filtrés par le motif générique : _.

II.2 Fonctions usuelles

Dans cette section nous allons écrire des fonctions qui existent dans le module `List`. Il est plus important de savoir les écrire que de se souvenir de leur nom.

Tête et queue (`List.hd` et `List.hd`) À l'aide du pattern-matching on peut écrire les déconstructions. On remarquera que l'on peut ne pas nommer les composantes inutilisées.

```
let tete liste =
  match liste with
  |[] -> failwith "Liste vide dans tete"
  |t::_ -> t;;
```

```
let queue liste =
  match liste with
  |[] -> failwith "Liste vide dans queue"
  |_::q -> q;;
```

Longueur (`List.length`) La longueur d'une liste s'écrit simplement aussi.

```
let longueur liste =
  match liste with
  |[] -> 0
  |t::q -> 1 + (longueur q);;
```

Retournement (`List.rev`)

Pour retourner une liste on place les élément de la liste en les lisant de la gauche vers la droite) dans une liste, au départ vide, ils seront donc écrits de la droite vers la gauche.

```
let retourne liste =
  let rec aux aFaire fait =
    match aFaire with
    |[] -> fait
    |t::q -> aux q (t::fait) in
  aux liste [];;
```

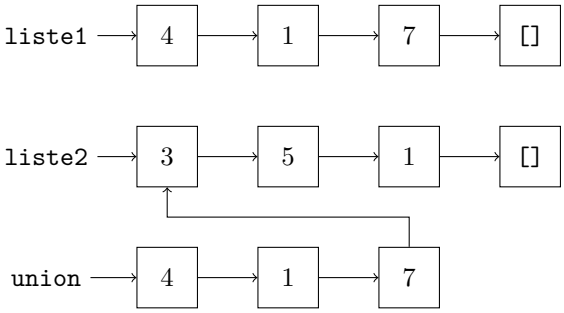
Appartenance (`List.mem`)

```
let rec appartient a liste =
  match liste with
  |[] -> false
  |t::q when t = a -> true
  |t::q -> appartient a q;;
```

Concaténation de listes (`@`)

Pour concaténer deux listes on va ajouter les éléments de la première à la seconde.

```
let rec union liste1 liste2 =
  match liste1 with
  |[] -> liste2
  |t::q -> t::(union q liste2);;
```



Le nombre d'opérations d'assemblage (`::`) est égal au nombre d'éléments de la première liste ; tous ces éléments sont copiés. Par contre les éléments de la seconde liste restent tels quels, ils sont communs à `liste2` et à `union liste1 liste2`.

La concaténation est aussi définie en OCaml par l'opérateur infixe `@` : `liste1 @ liste2`.

II.3 Récursivité terminale

Certaines fonctions ci-dessus ont un appel récursif qui consiste simplement à invoquer la fonction récursive ; c'est de cas de `appartient` et de `aux` dans `retourne`.
On parle dans ce cas de récursivité **terminale**.
C'est fonctions sont *facilement (?)* transformables en fonction non récursives avec un `while`.

```
let appartient a liste =
  let l = ref [] in
  while !l <> [] && List.hd !l <> a do l := List.tl !l done;
  !l <> [];;
  (* On renvoie true si on s'est arrêté parce qu'on a trouvé a *)
```

```
let retourne liste =
  let aFaire = ref liste in
  let fait = ref [] in
  while !aFaire <> [] do
     let a = List.hd !aFaire in
     aFaire := List.tl !aFaire;
     fait := a :: !fait done;
   !fait;;
```

OCaml (contrairement à Python) transformera immédiatement les fonctions récursives terminales en fonctions utilisant une boucle `while`. L'intérêt étant que la fonction n'utilise plus une pile de récursivité et qu'on évite ainsi la saturation de celle-ci (l'erreur `stack overflow`).
Lorsque la suite des appels récursifs risque d'être longue, il peut donc être utile d'utiliser une fonction avec une récursivité terminale. Par exemple le calcul (idiot) suivant

```
let rec accumuler_des_un n =
   match n with
   |0 -> 0
   |n -> 1 + accumuler_des_un (n - 1);;
```

engendre une erreur pour les grandes valeurs de n (de l'ordre de 10^5)

```
> accumuler_des_un 400000;;
Stack overflow during evaluation (looping recursion?).
```

La solution est alors d'écrire une fonction récursive terminale.

```
let accumuler_des_un n =
   let rec aux reste fait =
     match reste with
     |0 -> fait
     |n -> aux (reste - 1) (fait + 1)
   in aux n 0;;
```

La fonction `List.length` par exemple, utilise une récursivité terminale pour pouvoir gérer les (très) grandes listes.

```
let rec length_aux len = function
    [] -> len
  | _::l -> length_aux (len + 1) l;;

let length l = length_aux 0 l;;
```

La récursivité terminale construit les listes "à l'envers" ; c'est ce qui permet le retournement des listes dans `retourne`.

III Exercices

III.1 Gammes

Exercice 1 -
Écrire une fonction qui détermine si tous les termes d'une liste sont positifs ou nuls.

```
positive : int list -> int
```

Exercice 2 -
Écrire une fonction qui renvoie la liste des carrés des termes d'une liste.

```
carre : int list -> int list
```

Exercice 3 - Recherche avec critère : `List.exists`
Écrire une fonction qui teste si au moins un élément d'une liste vérifie un critère donné une fonction booléenne : `test`

```
existe : ('a -> bool) -> 'a list -> bool
```

Exercice 4 - Vérification totale : `List.forall`
Écrire une fonction qui teste si tous les éléments d'une liste vérifient un critère.

```
tous : ('a -> bool) -> 'a list -> bool
```

On n'est pas obligé de définir la fonction de test, par exemple la réponse à la question 1 peut s'écrire `let positive = tous (fun x -> x >= 0);;`.

Exercice 5 - Premier élément vérifiant un critère : `List.find`
Écrire une fonction qui renvoie le premier élément d'une liste qui vérifie un critère ; si aucun élément ne vérifie le critère la fonction doit renvoyer une erreur (`failwith`).

```
premier : ('a -> bool) -> 'a list -> 'a
```

La question précédente ne sera utilisée en pratique que si on a testé préalablement l'existence d'un élément vérifiant le critère ; on parcourt donc deux fois la liste. On peut simplifier les choses en répondant aux deux questions (existence et valeur d'un élément vérifiant le critère) en même temps grâce au type optionnel.

Exercice 6 - Filtrage : `List.filter`
Écrire une fonction qui renvoie la liste des éléments d'une liste qui vérifient un critère ; l'ordre des éléments dans la liste initiale doit être préservé.

```
filtrage : ('a -> bool) -> 'a list ->  'a list
```

Exercice 7 - Partition : `List.partition`
Améliorer le résultat précédent en renvoyant deux listes : la première contient les éléments de la liste qui vérifient le critère, la seconde contient ceux qui ne le vérifient pas, tout en conservant l'ordre.

```
separation : ('a -> bool) -> 'a list ->  'a list * 'a list
```

Exercice 8 - k-ième élément : `List.nth`
Écrire une fonction qui renvoie l'élément à la position k dans une liste, la tête de la liste étant à la position 0. Si l'indice est strictement négatif ou supérieur à la longueur de la liste, la fonction doit retourner une erreur.

```
k_ieme : 'a list -> int -> 'a list * 'a list
```

Exercice 9
Écrire une fonction qui sépare une liste en renvoyant deux listes ; la première contient les k premiers éléments et la seconde ceux qui restent. Si l'indice est strictement négatif ou supérieur à la longueur de la liste, la fonction doit retourner une erreur.

```
separer_rang : 'a list -> int -> 'a list * 'a list
```

Exercice 10
Abaissement de la fimension : `List.flatten`]Écrire une fonction qui transforme une liste de listes en une liste simple dont les éléments sont tous les éléments des listes composant la liste initiale.

```
aplatir : 'a list list -> 'a list
```

III.2 Listes de couples

Nous allons définir quelques fonctions sur des listes dont les éléments sont des couples. Cette structure intervient dans la notion de **dictionnaire** où les couples sont de la forme *(clé, valeur)* dans laquelle les clés sont nécessairement distinctes. On définit alors les opérations suivantes :

- recherche d'une `valeur` connaissant sa `clé` ;
- ajout d'un couple `(clé, valeur)` ;
- suppression d'un couple connaissant sa `clé`.

On notera `'a` le type des clés (ce sera souvent un entier) et `'b` celui des clés, typiquement un enregistrement.

Exercice 11 - Existence d'un couple : `List.mem_assoc`
Écrire une fonction qui teste s'il existe un couple dont la clé est donnée.

```
existe_cle : 'a -> ('a * 'b) list -> bool
```

Exercice 12 - Valeur associée : `List.assoc`
Écrire une fonction qui renvoie la valeur du couple dont la clé est donnée. S'il n'existe pas de tel couple la fonction doit renvoyer une erreur.

```
valeur : 'a -> ('a * 'b) list -> 'b
```

Exercice 13
Écrire une fonction qui ajoute un couple dans la liste. S'il existait déjà un couple avec la clé donnée en paramètre, la fonction devra simplement remplacer la valeur. Cette fonction renvoie une nouvelle liste.

```
ajouter : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list
```

Exercice 14
Enlever un couple : `List.remove_assoc`]
Écrire une fonction qui retire le couple de la liste dont la clé est donnée en paramètre. Si un tel couple n'existe pas, la fonction doit renvoyer une liste avec les mêmes éléments.

```
enlever : 'a -> ('a * 'b) list -> ('a * 'b) list
```

Exercice 15 - Séparation : `List.split`
Écrire une fonction qui sépare une liste de couples en deux listes.

```
dechirer : ('a * 'b) list -> 'a list * 'b list
```

Exercice 16 - Fusion : `List.combine`
Écrire une fonction qui combine deux listes de même longueur en une liste de couples. Si les deux listes n'ont pas la même longueur, la fonction doit lever une erreur.

```
combiner : 'a list -> 'b list -> ('a * 'b) list
```

III.3 Listes triées

Dans cette partie, on travaille avec des listes triées par ordre croissant.

Exercice 17
Écrire une fonction qui teste si une liste est bien triée par ordre croissant.

```
est_croissante : 'a list -> bool
```

Exercice 18
Écrire une fonction qui ajoute un élément à une liste triée en renvoyant une liste triée avec $n+1$ éléments si la liste initiale en contenait n.

```
inserer : 'a -> 'a list -> 'a list
```

Exercice 19
En déduire une fonction qui trie une liste.

```
tri_insertion : 'a list -> 'a list
```

Exercice 20
Écrire une fonction qui retire les éléments répétés d'une liste triée en n'en laissant qu'un. `[1; 1; 5; 8; 8; 8; 9; 10]` doit être transformée en `[1; 5; 8; 9; 10]`

```
nettoyer : : 'a list -> 'a list
```

Exercice 21
Écrire une fonction qui fusionne deux listes triées en une seule.

```
fusion : 'a list -> 'a list -> 'a list
```

III.4 Crible d'Érathostène

Le crible d'Érathostène permet de calculer les nombres premiers entre 2 et n.
Son principe est le suivant.

1. On écrit, dans l'ordre croissant, la suite des entiers de 2 à n.
2. On enlève de la liste tous les multiples entiers (à partir de $2k$) de chaque terme k (non enlevé) que l'on lit dans la liste.
3. Il ne reste alors que les nombres premiers.

Par exemple, pour $n = 20$, on calcule successivement

1. 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
2. 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, ~~18~~, 19, ~~20~~
3. 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~
4. 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~
5. 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~
6. 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~
7. 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~
8. 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~
9. 2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~

Les nombres premiers inférieurs à 20 sont 2, 3, 5, 7, 11, 13, 17, 19.

Exercice 22 - Énumération
Écrire une fonction `enum a b` qui renvoie la liste des entiers consécutifs de `a` à `b` dans l'ordre croissant, bornes comprises.

Exercice 23 - Enlever les multiples
Écrire une fonction `oterMult k liste` qui renvoie la liste des entiers de `liste` qui ne sont pas multiples de `k` dans le même ordre que dans la liste initiale.

Exercice 24 - Crible
Déduire des questions précédentes une fonction `premiers n` qui retourne la liste des nombres premiers inférieurs à n.
Combien d'opérations effectue-t-elle ?

Dans l'exemple on voit qu'il n'y a plus de termes à éliminer à partir de 5.
Ce phénomène est général : si $p^2 \geqslant n$, tous les multiples de p ont été retirés.

Exercice 25 - Amélioration
En déduire une modification de `premiers n`.
Combien d'opérations effectue-t-elle ?

III.5 Codes de Gray

Un code de Gray d'ordre n est une façon d'énumérer les entiers de 0 à $2^n - 1$ tel que le passage de l'écriture binaire du k-ème à celle du $(k+1)$-ème nombre se fait en ne changeant qu'un seul bit.

Exercice 26
Déterminer un code de Gray d'ordre 3.

On suppose que Γ est un code de Gray d'ordre n considéré comme une liste ℓ de 2^n chaînes de caractères de même longueur n dont les lettres sont 0 ou 1.

Exercice 27
Montrez que la liste obtenue en concaténant les mots de ℓ précédés de 0 puis les mots de la liste inverse de ℓ précédés de 1 est un code de Gray d'ordre $n+1$.

Exercice 28
En déduire une fonction construisant un code de Gray d'ordre n.

Les codes de Gray obtenus avec la construction qui vient d'être décrite sont dits réfléchis : on les note Γ_n. $g_n(k)$ est l'entier représenté par le terme à la position k de Γ_n pour $0 \leqslant k < 2^n$.

Exercice 29
Déterminer $g_n(0)$, $g_n(2^n - 1)$, $g_n(2^{n-1} - 1)$ et $g_n(2^{n-1})$.
Comparer $g_n(k)$ et 2^{n-1} pour $k < 2^{n-1}$ et $k \geqslant 2^{n-1}$.

Exercice 30
Comment passer du k-ième terme de Γ_n au $(k+1)$-ième ?

III.6 Division egyptienne

Pour effectuer la division de a par b la méthode utilisée dans l'Egypte antique consiste à calculer les termes b, $2b$, $4b$ ...jusqu'au dernier terme inférieur à a. On soustrait ensuite les termes de la forme $2^p b$ à partir du plus grand quand cela est possible : le terme restant est le reste. Le quotient est obtenu en ajoutant les puissance de 2 correspondant aux produits soustraits.
Exemple : pour diviser 2874 par 53 on calcule les valeurs

2^p	1	2	4	8	16	32
53.2^p	53	106	212	424	848	1696

$2874 - 1696 = 1178$, $1178 - 848 = 330$, on ne soustrait pas 424, $330 - 212 = 118$ puis $118 - 106 = 12$: le reste est 12. Le quotient est $32 + 16 + 4 + 2 = 54$.

Exercice 31
Déterminer la liste (décroissante) des $2^p.b$ inférieurs à a.

Exercice 32
Écrire une fonction `reste a b` qui calcule le reste de la division de a par b en utilisant cette méthode.

Exercice 33
Quel est le nombre de multiplications et de soustractions effectué ?

III.7 Sous-listes

Une sous-liste de la liste $m = [x_0; \ldots; x_{n-1}]$ est une liste obtenue en supprimant certains éléments de m (et en conservant l'ordre), c'est-à-dire une liste (éventuellement vide) de la forme $[x_{\sigma(0)}; \ldots; x_{\sigma(k-1)}]$ avec $0 \leq \sigma(0) < \cdots < \sigma(k-1) \leqslant n-1$.
Par exemple, $[0; 1; 2; 2; 3; 3]$ est une sous-liste de $[4; 0; 1; 2; 2; 2; 2; 3; 4; 3]$.

Exercice 34
Écrire une fonction `est_sous_liste` qui prend pour argument deux listes m_1 et m_2 et teste si m_1 est une sous-liste de m_2.

Étant donnée une liste m, on note $\Sigma(m)$ l'ensemble des sous-listes de la liste m.

Exercice 35
Soit `m = t::q` une liste non vide. Montrer que $\Sigma(m)$ est l'union de $\Sigma(q)$ et de l'ensemble des listes `t::r` où `r` décrit $\Sigma(q)$.

Exercice 36
En déduire une fonction `sous_listes` qui prend pour argument une liste m et calcul l'ensemble $\Sigma(m)$ (cet ensemble sera retourné sous forme d'une liste de listes, la même sous-liste pouvant éventuellement être répétée plusieurs fois).

III.8 Décomposition de Fibonacci

(F_n) est la suite de Fibonacci définie par $F_0 = 1$, $F_1 = 1$ et $F_{n+2} = F_n + F_{n+1}$ pour $n \geqslant 0$.
Pour k et m entiers, on note $k \gg m$ quand $k \geqslant m + 2$.
Une décomposition de $n \in \mathbb{N}$ dans la base de Fibonacci est la donnée de $r \in \mathbb{N}$ et de $(r+1)$ nombres de Fibonacci $(F_{i_r}, \ldots, F_{i_1}, F_{i_0})$ tels que

$n = \sum_{k=0}^{r} F_{i_k}$ avec $i_r \gg i_{r-1} \gg \cdots \gg i_1 \gg i_0 \geqslant 1$.

Exercice 37
Prouver que si $(F_{i_r}, \ldots, F_{i_1}, F_{i_0})$ est une décomposition de n alors $F_{i_r} \leqslant n < F_{i_r+1}$.

Exercice 38
Montrer que, pour tout $n \in \mathbb{N}^*$, il existe une unique décomposition de Fibonacci de n.

Exercice 39
Déterminer cette décomposition pour $n = 67$.

Exercice 40
Écrire une fonction `listeFibo : int -> int * int list` qui prend en argument un entier $n \geqslant 1$ et qui renvoie un couple formé de l'indice p et de la liste $(F_p, F_{p-1}, \ldots, F_1, F_0)$ avec p tel que $F_p \leqslant n < F_{p+1}$.

Exercice 41
Écrire une fonction `decompose : int -> int list` qui renvoie l'unique décomposition de Fibonacci de n.

Exercice 42
Écrire une fonction `somme1 : int -> int list -> int list` qui ajoute un nombre de Fibonacci F_p représenté par $p \geqslant 1$ a une décomposition de Fibonacci. Le résultat doit être sous la forme d'une décomposition de Fibonacci ne doit pas calculer l'entier associé à la liste.

Exercice 43
Écrire une fonction `somme : int list -> int list -> int list` qui calcule la somme de deux entiers représentés par leur décomposition de Fibonacci. Le résultat doit être sous la forme d'une décomposition de Fibonacci et on ne passera pas par les entiers

Solutions

Solution de l'exercice 1 -

```
let rec positive liste =
  match liste with
  |[] -> true
  |t::q -> (t >= 0) && (positive q);;
```

Solution de l'exercice 2 -

```
let rec carre liste =
  match liste with
  |[] -> []
  |t::q -> (t*t) :: (carre q);;
```

Solution de l'exercice 3 - Recherche avec critère : `List.exists`

```
let rec existe test liste =
  match liste with
  |[] -> false
  |t::q when test t -> true
  |t::q -> existe test q;;
```

Solution de l'exercice 4 - Vérification totale : `List.forall`

```
let rec tous test liste =
  match liste with
  |[] -> false
  |t::q -> (test t) && (tous test q);;
```

Solution de l'exercice 5 - Premier élément vérifiant un critère : `List.find`

```
let rec premier test liste =
  match liste with
  |[] -> failwith "Aucun élément ne vérifie le critère"
  |t::q when test t -> t
  |t::q -> premier test q;;
```

Solution de l'exercice 6 - Filtrage : `List.filter`

```
let rec filtrage test liste =
  match liste with
  |[] -> []
  |t::q when test t -> t :: (filtrage q)
  |t::q -> filtrage q;;
```

Si on veut utiliser une récursivité terminale on doit retourner le résultat

```
let filtrage test liste =
  let rec aux_flt reste trouves =
    match reste with
    |[] -> trouves
    |t::q when test t -> aux_flt q (t::trouves)
    |t::q  -> aux_flt q trouves in
  List.rev (aux_flt liste []);;
```

On peut retourner plus tôt

```
let filtrage test liste =
  let rec aux_flt reste trouves =
    match reste with
    |[] -> List.rev trouves
    |t::q when test t -> aux_flt q (t::trouves)
    |t::q  -> aux_flt q trouves in
   aux_flt liste [];;
```

Solution de l'exercice 7 - Partition : `List.partition`

```
let rec separation test liste =
  match liste with
  |[] -> [], []
  |t::q let l1, l2 = separation q in
        if test t then (t::l1), l2 else l1, (t::l2);;
```

Si on veut utiliser une récursivité terminale on doit retourner les résultats

```
let separation test liste =
  let rec aux_sep reste oui non =
    match reste with
    |[] -> List.rev oui, List.rev non
    |t::q when test t -> aux_sep q (t::oui) non
    |t::q  -> aux_sep q oui (t::non) in
  aux_sep liste [] [];;
```

Solution de l'exercice 8 - k-ième élément : `List.nth`
Une solution qui parcourt toute la liste même pour $k < 0$.

```
let rec k_ieme liste k =
  match list with
  |[] -> failwith "L'indice ne convient pas"
  |t::q when k = 0 -> t
  |t::q -> k_ieme (k-1) q;;
```

Si on veut faire plus efficace :

```
let rec k_ieme liste k =
  if k < 0
  then failwith "L'indice doit être positif"
  else match list with
       |[] -> failwith "L'indice est trop grand"
       |t::q when k = 0 -> t
       |t::q -> k_ieme q (k-1);;
```

Solution de l'exercice 9

```
let rec separer_rang liste k =
  if k < 0
  then failwith "L'indice doit être positif"
  else match k, liste with
       |0, _ -> [], liste
       |_, [] -> failwith "L'indice est trop grand"
       |t::q -> let l1, l2 = separer_rang q (k-1)
                in (t::l1), l2;;
```

Solution de l'exercice 10

```
let rec aplatir liste =
  match liste with
  |[] -> []
  |t::q -> t @ (aplatir q);;
```

Solution de l'exercice 11 - Existence d'un couple : `List.mem_assoc`

```
let rec existe_cle c liste =
  match liste with
  |[] -> false
  |(a, b)::q when a = c -> true
  |t::q -> existe_cle c q;;
```

Solution de l'exercice 12 - Valeur associée : `List.assoc`

```
let rec valeur c liste =
  match liste with
  |[] -> failwith "La clé n'apparaît pas dans la liste"
  |(a, b)::q when a = c -> b
  |t::q -> valeur c q;;
```

Solution de l'exercice 13

```
let rec ajouter c v liste =
  match liste with
  |[] -> [(c, v)]
  |(a, b)::q when a = c -> (c, v) :: q
  |t::q -> t::(ajouter c v q);;
```

Solution de l'exercice 14

```
let rec enlever c liste =
  match liste with
  |[] -> []
  |(a, b)::q when a = c -> q
  |t::q -> t::(enlever c q);;
```

Solution de l'exercice 15 - Séparation : `List.split`

```
let rec dechirer liste =
  match liste with
  |[] -> [], []
  |(a, b)::q -> let l1, l2 = dechirer q
                in (a::l1,  b::l2);;
```

Solution de l'exercice 16 - Fusion : `List.combine`

```
let rec combiner l1 l2 =
  match l1, l2 with
  |[], [] -> []
  |t1::q1, t2::q2 -> (t1, t2)::(combiner q1 q2)
  |_ -> failwith "Les deux listes n'ont pas la même longueur";;
```

Solution de l'exercice 17

```
let rec est croissante liste =
  match liste with
  |a::b::q -> (a <= b) && est_croissante b::q
  |_ -> true;;
```

Solution de l'exercice 18

```
let rec inserer a liste =
  match liste with
  |[] -> [a]
  |t::q when a <= t -> a::liste
  |t::q -> t :: (inserer a q);;
```

Solution de l'exercice 19

```
let rec tri_insertion =
  match liste with
  |[] -> []
  |t::q -> inserer t (tri_insertion q);;
```

Solution de l'exercice 20

```
let rec nettoyer liste =
  match liste with
  |a::b::q when a = b -> nettoyer b::q
  |a::b::q -> a :: (nettoyer b::q)
  |_ -> liste;;
```

Solution de l'exercice 21

```
let rec fusion l1 l2 =
  match l1,l2 with
  |[],_ -> l2
  |_,[] -> l1
  |t1::q1,t2::q2 -> if t1 < t2
                   then t1::(fusion q1 l2)
                   else t2::(fusion l1 q2);;
```

Solution de l'exercice 22 - Énumération

```
let rec enum a b =
  if a > b then []
           else a::(enum (a+1) b);;
```

Solution de l'exercice 23 - Enlever les multiples

```
let rec oterMult k liste =
  match liste with
  |[] -> []
  |t::q -> if t mod k = 0
           then oterMult k q
           else t::(oterMult k q);;
```

Solution de l'exercice 24 - Crible

```
let premier n =
  let rec elim liste =
    match liste with
    |[] -> []
    |t::q -> t::(elim (oterMult t q)) in
  elim (enum 2 n);;
```

À chaque nombre premier on parcourt la liste restante. Il y a au plus n nombre premiers et les listes sont de taille n au plus : on effectue au plus n^2 instructions.
N.B. On peut imaginer que cette majoration n'est pas optimale ; en fait le théorème de densité des nombres premiers dit que, si $\pi(n)$ est le nombre de premiers inférieurs à n alors $\pi(n) \sim \ln(n)$. On obtient ainsi une majoration en $n\ln(n)$.

Solution de l'exercice 25 - Amélioration

```
let premier n =
  let rec elim liste =
    match liste with
    |[] -> []
    |t::q -> if t*t <= n
             then t::(elim (oterMult t q))
             else liste in
  elim (enum 2 n);;
```

On effectue maintenant au plus $n^{3/2}$ instructions.

Solution de l'exercice 26
Les entiers de 0 à 7 ont pour développements binaires 000, 001, 010, 011, 100, 101, 110, 111. Un code de gray peut être 000, 010, 110, 100, 101, 111, 011, 001 (soit 0, 2, 6, 4, 5, 7, 3, 1).

Solution de l'exercice 27
On traduit l'énoncé : $a_0, a_1, \ldots, a_p$ est un code de Gray ($p = 2^n - 1$).
Alors $0a_0, 0a_1, \ldots, 0a_{p-1}, 0a_p, 1a_p, 1a_{p-1}, ldots, 1a_1, 1a_0$ est un code de Gray. En effet

- une seule lettre change entre $0a_i$ et $0a_{i+1}$ car une seule est changée entre a_i et a_{i+1},
- de même une seule lettre change entre $1a_{i+1}$ et $1a_i$,
- il reste $0a_p$ et $1a_p$ entre lesquels seule la première lettre change.

Solution de l'exercice 28
On décompose en fonctions simples les étapes.

```
let retourner liste =
  let rec aux l1 l2 =
    match l1 with
    |[] -> l2
    |t::q -> aux q (t::l2) in
  aux liste [];;

let ajouter char liste =
  map (fun x -> char^x) liste;;

let rec gray n =
  if n = 0
  then [""]
  else let  l = gray (n-1) in
       (ajouter "0" l)@(ajouter "1" (retourner l));;
```

Solution de l'exercice 29
On a toujours $g_n(0) = 0$.
$g_n(2^n - 1)$ s'obtient en ajoutant un 1 en binaire devant $g_n(0)$ à la place $n - 1$: on obtient 2^{n-1}.
$g_n(2^{n-1} - 1) = g_{n-1}(2^{n-1} - 1) = 2^{n-2}$
$g_n(2^{n-1})$ s'obtient en ajoutant un 1 en binaire devant $g_n(2^{n-1} - 1)$ à la place $n - 1$: on obtient $2^{n-1} + 2^{n-2} = 3.2^{n-2}$.
En raison de la construction par symétrie on a $g_n(k) < 2^{n-1}$ pour $k < 2^{n-1}$ et $g_n(k) \geqslant 2^{n-1}$ pour $k \geqslant 2^{n-1}$.

Solution de l'exercice 30
On utilise la question précédente.
Le successeur de 2^{n-2} est 3.2^{n-2} et celui de 2^{n-1} et 0.
Pour $k < 2^{n-1}$ (et $k \neq 2^{n-2}$ le successeur de k dans Γ_n est son successeur dans Γ_{n-1}.
En raison de la symétrie, pour $k > 2^{n-1}$ le successeur de k est $2^{n-1} + k'$ où k' est le prédécesseur de k_2^{n-1} dans Γ_{n-1}.
On va donc écrire deux fonctions récursives simultanées.

```
let rec grayPlus k n =
  if n = 1
  then 1-k
  else (let p = puiss2 (n-2) in
        if k = 2*p
        then 0
        else if k = p
             then 3*p
             else if k < 2*p
                  then grayPlus k (n-1)
                  else 2*p + grayMoins (k-2*p) (n-1))
and grayMoins k n =
  if n = 1
  then 1-k
  else (let q = puiss2 (n-2) in
        if k = 0
        then 2*q
        else if k = 3*q
             then q
             else if k < 2*q
                  then grayMoins k (n-1)
                  else 2*q + grayPlus (k-2*q) (n-1));;
```

Solution de l'exercice 31

Si la liste décroissante de ces termes est $[2^k b; 2^{k-1}b; \ldots; 2b; b]$ alors la liste des termes pour $a/2$ est $[2^{k-1}b; \ldots; 2b; b]$. On a un algorithme récursif.

```
let rec puiss2b a b =
  if a < b
  then []
  else match (puiss2b (a/2) b) with
       |[] -> [b]
       |t::q -> (2*t)::(t::q);;
```

Solution de l'exercice 32

```
let reste a b =
 let rec aux l petit_a =
    match l with
    |[] -> petit_a
    |t::q -> if t <= petit_a
             then aux q (petit_a - t)
             else aux q petit_a in
  aux (puiss2b a b) a;;
```

Solution de l'exercice 33

Le nombre d'opération est un $\text{‰}\lambda$ où λ est la longueur de la liste `puiss2b a b` que ce soit pour le calcul de cette liste ou pour son exploitation dans la fonction `reste`.
Or le nombre de termes de la forme $b2^k$ majorés par a est $p+1$ si on a $2^p < \frac{a}{b} \leqslant 2^{p+1}$.
La complexité de `reste` est donc un $\text{‰}\ln(b/a) = \text{‰}\ln(b)$.

Solution de l'exercice 34

```
let rec est_sous_liste m1 m2 =
  match m1, m2 with
  |[], _ -> true
  |_, [] -> false
  |t1 :: q1, t2 :: q2 -> if t1 = t2
                         then est_sous_liste q1 q2
                         else est_sous_liste m1 q2;;
```

Solution de l'exercice 35
On sépare les sous-listes selon qu'elles contiennent t ou pas.

Solution de l'exercice 36

```
let rec sous_listes = function
  |[] -> [[]]
  | t :: q -> let e = sous_listes q in
              e @ (map (fun m -> t :: m) e);;
```

Solution de l'exercice 37
L'inégalité $F_{i_r} \leqslant n$ est évidente.
On montre ensuite par récurrence sur r que si on a $i_r \gg i_{r-1} \gg \cdots \gg i_1 \gg i_0 \geqslant 1$ alors $\sum_{k=0}^{r} F_{i_k} < F_{i_r+1}$.
C'est vrai pour $r = 0$ car $i \geqslant 1$ implique $F_i < F_{i+1}$.
Si la propriété est vraie pour r alors $i_{r+1} \gg i_r \gg i_{r-1} \gg \cdots \gg i_1 \gg i_0 \geqslant 1$ implique $i_{r+1} \geqslant i_r + 2$ et $\sum_{k=0}^{r} F_{i_k} < F_{i_r+1} \leqslant F_{i_{r+1}-1}$ puis $\sum_{k=0}^{r+1} F_{i_k} < F_{i_{r+1}-1} + F_{i_{r+1}} = F_{i_{r+1}+1}$.

Solution de l'exercice 38
On procède par récurrence.
Pour $n = 1$, la seule décomposition possible est (F_1).
On suppose que tout entier k avec $1 \leqslant k < n$ avec $n \geqslant 2$ admet une décomposition unique.
D'après la question précédente, si $n \geqslant 2$ admet une décomposition le premier terme doit être p tel que $F_p \leqslant n < F_{p+1}$.
On a $2 = F_2 \leqslant n < F_{p+1}$ donc $p + 1 > 2$ puis $p \geqslant 2$.
On a alors $n - F_p < F_{p+1} - F_p = F_{p-1}$.
Si $n = F_p$ alors (F_p) est une décomposition et c'est la seule car elle doit contenir F_p.
Si $n > F_p$ alors $n - F_p$ admet une décomposition unique $(F_{i_r}, F_{i_{r-1}}, \ldots, F_{i_0})$. D'après la question précédente on a $F_{i_r} \leqslant n - F_p < F_{i_r+1}$ et on a vu qu'on a $n - F_p < F_{p-1}$. On a donc $i_r < p - 1$ donc $p \gg i_r$.
Ainsi $(F_p, F_{i_r}, F_{i_{r-1}}, \ldots, F_{i_0})$ est une décomposition de $F_p + n - F - p = n$ et c'est la seule en raison de l'unicité du premier terme et de l'unicité de la décomposition de $n - F_p$.

Solution de l'exercice 39
On a $F_9 = 55 \leqslant 67 < 89 = F_{10}$, $F_5 = 8 \leqslant 67 - 55 = 12 < 13 = F_6$, $F_3 = 3 \leqslant 12 - 8 = 4 < 5 = F_4$ et $4 - 3 = 1 - F_1$ donc la décomposition de 67 est (55,8,3,1).

Solution de l'exercice 40

```
let suivant liste =
  match liste with
  |t::s::q -> t+s
  |_ -> failwith "erreur dans suivant" ;;

let listeFibo n =
  let rec construction liste =
    let suiv = suivant liste in
    if suiv <= n
    then construction (suiv::liste)
    else liste in
  construction [1; 1];;
```

Solution de l'exercice 41
On applique l'algorithme :

```
let rec decompose n =
  if n = 0
  then []
  else let k = List.hd (listeFibo n) in
       k::(decompose (n - k));;
```

La fonction ci-dessus calcule la suite de Fibonacci pour chaque nouvelle valeur.
On peut améliorer avec

```
let decompose n =
  let rec aux k liste =
    match liste with
    |[] -> []
    |t::q -> if t > k then aux k q else t::(aux (k-t)) q in
    aux n (listeFibo n);;
```

Solution de l'exercice 42
On ajoute récursivement à une représentation en différenciant les cas en fonction du premier terme de la représentation.
Comme on va utiliser $p-2$ il faut gérer le cas $p \leqslant 1$.
Pour prouver que le résultat est une représentation n utilise la propriété, que l'on montre par récurrence, que si le premier terme de la représentation est majoré par p alors le premier terme du résultat est majoré par $p+1$.

```
let rec somme1 p listeF =
  match listeF with
  |[] -> [p]
  |q::reste when q < p-1 -> p::listeF
  |q::reste when q = p-1 -> (p+1)::reste
  |q::reste when q = p && p= 1 -> [1]
  |q::reste when q = p && p= 2 -> [3; 1]
  |q::reste when q = p -> (p+1)::(somme1 (p-2) reste)
  |q::reste -> somme1 q (somme1 p reste);;
```

Solution de l'exercice 43
On ajoute récursivement chaque terme de la liste.

```
let rec somme liste1 liste2 =
  match liste1 with
  |[] -> liste2
  |p::reste -> somme1 p (somme reste liste2)::
```