Consignes. Les calculatrices sont interdites. Numérotez vos feuilles et faites apparaître les questions dans l'ordre du sujet. Si vous repérez une erreur d'énoncé, signalez-la sur votre copie et poursuivez votre composition.

Rappel. Pour manipuler des listes chaînées en OCaml, il est naturel d'utiliser des fonctions récursives et non des fonctions itératives. Si c'est pertinent vous pouvez écrire des fonctions itératives, mais dans la plupart des questions l'utilisation de fonctions itératives à la place de fonctions récursives sera sanctionnée.

1 Introduction

Dans tout le sujet, on s'intéresse à la notion de ***chaînes d'additions***.

Définition informelle. Une chaîne d'additions est un uplet u d'entiers strictement croissants construit à l'aide de deux règles :

(règle 1) Initialement, u contient uniquement l'entier 1.

(règle 2) À chaque étape, on choisit deux entiers x, y dans u et on ajoute $x+y$ à la fin de u.

Par exemple, le uplet $u = (1,2,3,6,8,9)$ est une chaîne d'additions :

$$\begin{aligned}
(1) &\rightsquigarrow (1,2) && \text{en appliquant la règle 2 avec } x=1 \text{ et } y=1\\
&\rightsquigarrow (1,2,3) && \text{en appliquant la règle 2 avec } x=1 \text{ et } y=2\\
&\rightsquigarrow (1,2,3,6) && \text{en appliquant la règle 2 avec } x=3 \text{ et } y=3\\
&\rightsquigarrow (1,2,3,6,8) && \text{en appliquant la règle 2 avec } x=2 \text{ et } y=6\\
&\rightsquigarrow (1,2,3,6,8,9) && \text{en appliquant la règle 2 avec } x=1 \text{ et } y=8
\end{aligned}$$

Définition formelle. Soit $s \in \mathbb{N}$ et $u = (u_0, u_1, \ldots, u_s) \in (\mathbb{N}^*)^{s+1}$ un $(s+1)$-uplet d'entiers strictement positifs. On dit que u est une ***chaîne d'additions*** s'il vérifie deux propriétés :

(i) $u_0 = 1$ et $u_0 < u_1 < u_2 < \ldots < u_s$.

(ii) $\forall i \in [\![1\,;s]\!], \exists (j,k) \in [\![0\,;i-1]\!]^2, u_i = u_j + u_k$.

Dans tout le sujet, l'entier s sera appelé la ***longueur*** de u et u_s sera appelé le ***dernier élément*** de u. Ainsi :

→ $(1,2,3,6,8,9)$ est une chaîne d'additions de longueur 5 dont le dernier élément est 9.

→ $(1,2,3,4,5,7,14)$ est une chaîne d'additions de longueur 6 dont le dernier élément est 14.

→ $(1,2,4,8,16,32,64,128)$ est une chaîne d'additions de longueur 7 dont le dernier élément est 128.

En revanche, le uplet $(1,2,4,9,18)$ n'est pas une chaîne d'additions, car le point (ii) n'est pas vérifié par $i=3$ (pour lequel $u_i = 9$).

1. Dans cette question, les réponses peuvent être données sans justification.
   (a) Pour chacun des uplets suivants, déterminer s'il s'agit d'une chaîne d'additions.
   $$u_1 = (1,2,4,8,16,20,21) \qquad u_2 = (1,2,5,10,21) \qquad u_3 = (1,2,3,4,5,10,15,20,21)$$
   (b) Donner une chaîne d'additions de longueur 10 dont le dernier élément est 1024.
   (c) Donner une chaîne d'additions dont le dernier élément est 77. On fera en sorte que la chaîne soit la plus courte possible.
2. Écrire une fonction `dernier: int list -> int` qui renvoie le dernier élément de la liste en entrée.

En OCaml, une chaîne d'additions $(u_0, u_1, \ldots, u_s)$ est représentée par la liste chaînée `[u0; u1; ...; us]`. On définit donc le type :

```
type chAdd = int list;;
```

Lorsqu'une fonction prend en entrée un objet de type `chAdd`, vous pouvez supposer sans le vérifier que les propriétés (i) et (ii) évoquées ci-dessus sont vraies. De plus, si une fonction doit renvoyer un objet de type `chAdd`, alors vous devez faire en sorte que cet objet vérifie les propriétés (i) et (ii).

3. Écrire une fonction `ex: int -> chAdd` qui prend en entrée un entier n et renvoie une chaîne d'additions (de votre choix) dont le dernier élément est n. Si $n \leqslant 0$, votre fonction déclenchera une erreur.

Soit $n \in \mathbb{N}^*$. Jusqu'à la fin du sujet, notre objectif est d'étudier différentes stratégies pour construire une chaîne d'additions dont le dernier élément est n.

2 Stratégies simples pour générer des chaînes d'additions

2.1 Stratégie binaire

La première stratégie que nous allons employer s'appelle la "stratégie binaire". L'idée est de diviser le nombre considéré par 2 (avec une division entière) jusqu'à obtenir 1. Par exemple avec $n = 179$:

$$179 \rightsquigarrow 89 \rightsquigarrow 44 \rightsquigarrow 22 \rightsquigarrow 11 \rightsquigarrow 5 \rightsquigarrow 2 \rightsquigarrow 1$$

En remontant ces calculs, on construit la chaîne d'additions $(1, 2, 4, 5, 10, 11, 22, 44, 88, 89, 178, 179)$:

$(1) \rightsquigarrow (1, 2)$	on ajoute 2 car 2 est pair
$\rightsquigarrow (1, 2, 4, 5)$	on ajoute 4 et 5 car 5 est impair
$\rightsquigarrow (1, 2, 4, 5, 10, 11)$	on ajoute 10 et 11 car 11 est impair
$\rightsquigarrow (1, 2, 4, 5, 10, 11, 22)$	on ajoute 22 car 22 est pair
$\rightsquigarrow (1, 2, 4, 5, 10, 11, 22, 44)$	on ajoute 44 car 44 est pair
$\rightsquigarrow (1, 2, 4, 5, 10, 11, 22, 44, 88, 89)$	on ajoute 88 et 89 car 89 est impair
$\rightsquigarrow (1, 2, 4, 5, 10, 11, 22, 44, 88, 89, 178, 179)$	on ajoute 178 et 179 car 179 est impair

4. Sans le justifier, donner la chaîne d'additions obtenue en appliquant la stratégie binaire à $n = 153$.
5. Écrire une fonction `bin: int -> chAdd` qui prend en entrée un entier $n \in \mathbb{N}^*$ et renvoie une chaîne d'additions dont le dernier élément est n. Vous devez utiliser la stratégie binaire décrite ci-dessus.

2.2 Stratégie force-brute

Soit $n \in \mathbb{N}^*$. La stratégie par force-brute consiste à générer toutes les chaînes d'additions se terminant par n, afin de trouver celle de longueur minimale.

Dans la suite, on manipulera des listes d'entiers (sans doublon) triées par ordre strictement croissant :

```
(* Listes triées sans doublon *)
type ens = int list;;
```

Lorsqu'une fonction prend en entrée un objet de type `ens`, vous pouvez supposer sans le vérifier qu'il s'agit d'une liste d'entiers sans doublon triée par ordre croissant. De plus, si une fonction doit renvoyer un objet de type `ens`, alors vous devez faire en sorte que la liste soit triée par ordre croissant et ne contienne pas de doublon.

6. Écrire une fonction `concat: ens -> ens -> ens` qui prend en entrée deux listes « `li1: ens` », « `li2: ens` » et renvoie une liste `li3` triée sans doublon contenant les mêmes éléments que `li1 @ li2`. Votre fonction devra être de complexité $\mathcal{O}(n)$ où n est la taille de `li3`.
7. (a) Écrire une fonction `som1: int -> ens -> ens` qui prend en entrée un entier x ainsi qu'une liste « `li: ens` » et renvoie une liste triée sans doublon contenant tous les entiers de la forme $x + y$ où y est un élément de `li`. Par exemple, « `som1 5 [1; 5; 9; 15]` » vaut `[6; 10; 14; 20]`.
   (b) Écrire une fonction `som2: ens -> ens` qui prend en entrée une liste « `li: ens` » et renvoie une liste triée sans doublon contenant tous les entiers de la forme $x + y$ où x et y sont des éléments de `li`. Par exemple, « `som2 [1; 2; 3; 4; 8; 16]` » vaut :

   `[2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 16; 17; 18; 19; 20; 24; 32]`
8. Écrire une fonction `cut: ens -> int -> int -> ens` qui prend en entrée une liste « `li: ens` » ainsi que deux entiers $(e_{\min}, e_{\max})$, et renvoie une liste triée sans doublon contenant tous les éléments x de `li` vérifiant $e_{\min} < x \leqslant e_{\max}$.

Soient $n_0 \in \mathbb{N}^*$ et $s \in \mathbb{N}$ deux entiers. On note $E_{n_0,s}$ l'ensemble de toutes les chaînes d'additions de longueur s dont tous les éléments sont inférieurs ou égaux à n_0. Par exemple pour $n_0 = 7$:

$$E_{7,0} = \Big\{(1)\Big\} \qquad E_{7,1} = \Big\{(1,2)\Big\} \qquad E_{7,2} = \Big\{(1,2,3),(1,2,4)\Big\}$$

$$E_{7,3} = \Big\{(1,2,3,4),(1,2,3,5),(1,2,3,6),(1,2,4,5),(1,2,4,6)\Big\}$$

$$E_{7,4} = \Big\{(1,2,3,4,5),(1,2,3,4,6),(1,2,3,4,7),(1,2,3,5,6),(1,2,3,5,7),(1,2,3,6,7),(1,2,4,5,6),(1,2,4,5,7),(1,2,4,6,7)\Big\}$$

$$E_{7,5} = \Big\{((1,2,3,4,5,6),(1,2,3,4,5,7),(1,2,3,4,6,7),(1,2,3,5,6,7),(1,2,4,5,6,7))\Big\} \qquad E_{7,6} = \Big\{(1,2,3,4,5,6,7)\Big\}$$

$$E_{7,7} = \Big\{\Big\}$$

Notre but est de construire un tableau « `tab: int array` » de taille $n_0 + 1$ tel que pour tout $n \in [\![\, 1\,;n_0\,]\!]$, l'élément d'indice n de `tab` est la longueur minimale d'une chaîne d'additions dont le dernier élément est n. Par exemple :

Pour $n_0 = 7$: `tab = [|-1; 0; 1; 2; 2; 3; 3; 4|]`
Pour $n_0 = 15$: `tab = [|-1; 0; 1; 2; 2; 3; 3; 4; 3; 4; 4; 5; 4; 5; 5; 5|]`

L'entier `tab.(0)` n'a pas de signification, vous pouvez le choisir arbitrairement.

9. Écrire une fonction `ext: chAdd -> int -> chAdd list` qui prend en entrée une chaîne d'additions u ainsi qu'un entier n_0, et renvoie la liste de toutes les chaînes d'additions qu'on peut obtenir en concaténant un nouvel entier à u. On pourra supposer sans le vérifier que les éléments de u sont inférieurs ou égaux à n_0. De plus, vous devez faire en sorte que les entiers présents dans la liste renvoyée soient tous inférieurs ou égaux à n_0.
   Par exemple, « `ext [1; 2; 4; 8; 10; 12] 20` » vaut :

```
[[1; 2; 4; 8; 10; 12; 13]; [1; 2; 4; 8; 10; 12; 14];
 [1; 2; 4; 8; 10; 12; 16]; [1; 2; 4; 8; 10; 12; 18]; [1; 2; 4; 8; 10; 12; 20]]
```

10. Soit $n_0 \in \mathbb{N}^*$ et $s \in \mathbb{N}$. On note `li1` la liste contenant les éléments de $E_{n_0,s}$ et `li2` la liste contenant ceux de $E_{n_0,s+1}$. Écrire une fonction `maj: chAdd list -> int -> chAdd list` qui prend en entrée `li1` ainsi que n_0, et renvoie `li2`.
11. Écrire une fonction `opti: int -> int array` qui prend en entrée l'entier n_0 et renvoie le tableau `tab` décrit ci-dessus.
12. Écrire une fonction `bin_opti: int -> int list` qui prend en entrée $n_0 \in \mathbb{N}^*$ et renvoie la liste de tous les entiers $k \in [\![1 ; n_0]\!]$ tels que la fonction `bin` (partie 2.1) appelée sur k renvoie une chaîne d'additions de longueur minimale. Par exemple, pour $n_0 = 100$ on obtient :

```
[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 16; 17; 18; 19; 20; 21; 22;
 24; 25; 26; 28; 29; 32; 33; 34; 35; 36; 37; 38; 40; 41; 42; 44; 48; 49; 50;
 52; 53; 56; 57; 58; 64; 65; 66; 67; 68; 69; 70; 71; 72; 73; 74; 76; 80; 81;
 82; 84; 88; 89; 96; 97; 98; 100]
```

3 Stratégies élaborées pour générer des chaînes d'additions

3.1 Opérations sur les chaînes d'additions

Dans cette partie, on définit deux opérations sur les chaînes d'additions.

Opération $\oplus$. Soit $u = (u_0, u_1, \ldots, u_s)$ une chaîne d'additions de longueur s et $n \in \mathbb{N}^*$ un entier strictement positif. On note $u \oplus n$ le $(s+2)$-uplet :

$$u \oplus n = (u_0, u_1, \ldots, u_s, u_s + n).$$

Remarquons que si $n \in \{u_0, u_1, \ldots, u_s\}$, alors $u \oplus n$ est une chaîne d'additions.

13. Écrire une fonction `mem: int -> chAdd -> bool` qui prend en entrée un entier n ainsi qu'une chaîne d'additions u, et indique si n apparaît dans u. Autrement dit, votre fonction doit renvoyer la même valeur que `List.mem`, mais vous n'avez pas le droit d'utiliser `List.mem`.
14. Écrire une fonction `add: chAdd -> int -> chAdd` qui prend en entrée une chaîne d'additions u ainsi qu'un entier n, et renvoie $u \oplus n$. Si n n'apparaît pas dans u, votre fonction déclenchera une erreur.

Opération $\otimes$. Soient $u = (u_0, u_1, u_2, \ldots, u_s)$ et $v = (v_0, v_1, v_2, \ldots, v_t)$ deux chaînes d'additions de longueurs respectives s et t. On note $u \otimes v$ la chaîne d'additions de longueur $s + t$ définie par :

$$u \otimes v = (u_0, u_1, u_2, \ldots, u_s, u_s v_1, u_s v_2, \ldots, u_s v_t)$$

Par exemple : $(1, 2, 4, 6, 12) \otimes (1, 2, 3, 5, 6) = (1, 2, 4, 6, 12, 24, 36, 60, 72)$.

15. Écrire une fonction `prod: chAdd -> chAdd -> chAdd` qui prend en entrée deux chaînes d'additions u, v, et renvoie $u \otimes v$.

3.2 Implémentation des stratégies

16. Écrire une fonction `pow2: int -> bool` qui prend en entrée un entier $n \in \mathbb{N}^*$ et renvoie `true` s'il existe un entier $k \in \mathbb{N}$ tel que $n = 2^k$, et `false` sinon.
17. Écrire une fonction `mini: chAdd list -> chAdd` qui prend en entrée `li` une liste de chaînes d'additions, et renvoie l'élément de `li` de longueur minimale. Si `li` est vide, votre fonction déclenchera une erreur.

18. Écrire une fonction `map : ('a -> 'b) -> 'a list -> 'b list` qui prend en entrée une fonction « `f: 'a -> 'b` » ainsi qu'une liste `[a1; a2; ...; an]` et renvoie la liste `[f a1; f a2; ...; f an]`. Autrement dit, votre fonction doit renvoyer la même valeur que `List.map`, mais vous n'avez pas le droit d'utiliser `List.map`.

On va maintenant décrire un ensemble de stratégies pour générer des chaînes d'additions. Soit « `strat: int -> int list` » une fonction telle que pour tout $n \geqslant 5$, `strat(n)` est une liste non vide d'éléments de $[\![2 ; n-1]\!]$. Voici une description informelle de la procédure que nous allons utiliser pour générer une chaîne d'additions dont le dernier élément est $n \geqslant 5$:

- On parcourt tous les éléments `m` de la liste `strat(n)`.
- Pour chaque `m`, on construit une chaîne d'additions contenant `m` et dont le dernier élément est `n`. Pour cela, on va utiliser des appels récursifs ainsi que les opérations $\oplus$ et $\otimes$ de la partie 3.1.
- Parmi toutes les chaînes d'additions générées dans le point précédent, on renvoie celle de longueur minimale.

À titre d'exemple, la stratégie binaire de la partie 2.1 correspond à la fonction :

```
let strat_bin (n: int) =
  [n/2];;
```

En effet, comme la liste renvoyée par la fonction `strat_bin` contient un seul élément, cela impose que la chaîne d'additions générée contienne `n/2`.
Plus concrètement, on s'intéresse à la fonction « `chaine : (int -> int list) -> int -> chAdd` » :

```
(* Hypothèse: n1 <= n2 *)
let rec chaine0 (strat: int -> int list) (n1: int) (n2: int): chAdd =
  match n1 with
  | 1 -> if pow2 n2 || n2 = 3 then bin n2 else begin
              let li = strat n2 in
              let li = map (fun n3 -> chaine0 strat n3 n2) li in
              mini li
            end
  | _ -> let r = n2 mod n1 in
         let q = n2 / n1 in
         if r = 0 then prod (chaine0 strat 1 n1) (chaine0 strat 1 q)
         else add (prod (chaine0 strat r n1) (chaine0 strat 1 q)) r;;

let chaine (strat: int -> int list) (n: int): chAdd = chaine0 strat 1 n;;
```

On pourra vérifier que pour tout $n \in \mathbb{N}^*$, les appels à « `bin n` » et « `chaine strat_bin n` » renvoient la même chaîne d'additions.

Stratégie co-binaire. La stratégie co-binaire consiste à utiliser la fonction :

```
let strat_coBin (n: int) =
  [(n+1)/2];;
```

19. Que vaut « `chaine strat_coBin 179` » ? On attend une justification détaillée.

Stratégie des facteurs. La fonction « `strat_facteurs: int -> int list` » est telle que pour tout $n \geqslant 5$:

$$\texttt{strat\_facteurs(n)} = \begin{cases} \texttt{[n-1]} \text{ si } n \text{ est un nombre premier.} \\ \texttt{[q, n-1]} \text{ sinon, où } q \text{ est le plus petit nombre premier qui divise } n. \end{cases}$$

20. Écrire la fonction `strat_facteurs`. On pourra supposer sans le vérifier que $n \geqslant 5$.

Stratégie totale. La fonction « `strat_totale: int -> int list` » est telle que :

$$\forall \texttt{n} \geqslant 5 : \texttt{strat\_totale(n) = [2; 3; ...; n-1]}.$$

21. Écrire la fonction `strat_totale`. On pourra supposer sans le vérifier que $\texttt{n} \geqslant 5$.

Stratégie dyadique. La fonction « `strat_dyadique: int -> int list` » prend en entrée un entier $n \geqslant 5$ et renvoie une liste contenant une seule fois chaque élément de l'ensemble :

$$[\![2 ; n-1]\!] \cap \left\{ \left\lfloor \frac{n}{2^j} \right\rfloor , j \in \mathbb{N}^* \right\}$$

22. Écrire la fonction `strat_dyadique`. On pourra supposer sans le vérifier que $n \geqslant 5$.

Stratégie de Fermat. La fonction « `strat_fermat: int -> int list` » prend en entrée un entier $n \geqslant 5$ et renvoie une liste contenant une seule fois chaque élément de l'ensemble :

$$[\![2 ; n-1]\!] \cap \left\{ \left\lfloor \frac{n}{2^{2^j}} \right\rfloor , j \in \mathbb{N} \right\}$$

23. Écrire la fonction `strat_fermat`. On pourra supposer sans le vérifier que $n \geqslant 5$.

Stratégie dichotomique. La fonction « `strat_dicho: int -> int list` » prend en entrée un entier $n \geqslant 5$ et renvoie la liste dont le seul élément est :

$$\left\lfloor \frac{n}{2^{\lceil \lambda(n)/2 \rceil}} \right\rfloor$$

où $\lambda(n) \in \mathbb{N}$ est l'entier tel que $\lambda(n)+1$ est le nombre de bits dans la représentation en base 2 de n.

24. Écrire la fonction `strat_dicho`. On pourra supposer sans le vérifier que $n \geqslant 5$.

3.3 Comparaison des différentes stratégies

25. Écrire une fonction `perf: (int -> int list) -> int -> int` qui prend en entrée une fonction « `strat: int -> int list` » ainsi qu'un entier $\texttt{n}_0 \in \mathbb{N}^*$, et renvoie la somme des longueurs des chaînes d'additions « `chaine strat n` » où `n` parcourt l'ensemble $[\![1 ; \texttt{n}_0]\!]$. Par exemple :

	`strat_bin`	`strat_coBin`	`strat_facteurs`	`strat_totale`	`strat_dyadique`	`strat_fermat`	`strat_dicho`
$\texttt{n}_0 = 100$	699	699	674	664	664	668	671
$\texttt{n}_0 = 1000$	11925	11925	11088	10821	10837	10927	11064