A2024 – INFO II MPI

**ÉCOLE DES PONTS PARISTECH,
ISAE-SUPAERO, ENSTA PARIS,
TÉLÉCOM PARIS, MINES PARIS,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS,
CHIMIE PARISTECH - PSL.**

**Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle International).**

CONCOURS 2024

DEUXIÈME ÉPREUVE D'INFORMATIQUE

Durée de l'épreuve : 4 heures

L'usage de la calculatrice et de tout dispositif électronique est interdit.

Cette épreuve concerne uniquement las candidats de la filière MPI.

L'énoncé de cette épreuve comporte 12 pages de texte.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Préliminaires

L'épreuve est composée d'un problème unique, comportant 36 questions. Le problème est divisé en quatre sections. Dans la première section (page 1), nous introduisons une forme paresseuse de listes, les *flots de données*, et construisons quelques fonctions de base utiles tout au long du sujet. Dans la deuxième section (page 4), nous nous intéressons à un *algorithme de détermination à la volée du cardinal* d'un flot, dont l'inspiration est probabiliste. Dans la troisième section (page 7), nous nous appuyons sur des raisonnements de logique pour analyser le *chiffrement par flot*. Dans la quatrième section (page 9), nous traitons quelques questions de *combinatoire énumérative* à l'aide d'une grammaire ou simplement d'un procédé manuel de dérécursification.

Les sections 2, 3 et 4 peuvent être vues comme des exercices indépendants qui ne dépendent que de la première section.

Dans tout l'énoncé, un même identificateur écrit dans deux polices de caractères différentes désigne la même entité, mais du point de vue mathématique pour la police en italique (par exemple n) et du point de vue informatique pour celle en romain avec espacement fixe (par exemple `n`).

Des rappels portant sur des extraits du manuel de documentation de OCaml et sur les règles de la déduction naturelle sont fournis en annexe.

Travail attendu

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat.

Il faudra coder des fonctions à l'aide du langage de programmation OCaml exclusivement, en reprenant l'en-tête de fonctions fourni par le sujet, sans s'obliger à recopier la déclaration des types. Il est permis d'utiliser la totalité du langage OCaml mais il est recommandé de s'en tenir aux fonctions les plus courantes afin de rester compréhensible. Quand l'énoncé demande de coder une fonction, sauf demande explicite, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté et de la concision des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle.

1. Flots de données

1.1. Motivation

En langage OCaml, le type natif `'a list` est défini comme point fixe de l'équation

```
1.  type 'a list = [] | (::) of 'a * 'a list
```

Ce type permet de définir des listes finies tout comme certaines listes infinies, ainsi que l'expose l'exemple suivant, qui est parfaitement licite :

```
2.  let list_finite = [ 1; 2; 3 ]        (* liste finie a 3 elements *)
3.  let rec list_ones = 1 :: list_ones   (* liste infinie toute a 1 *)
```

☐ 1 – Rendre compte, sous la forme d'un croquis constitué de maillons chaînés entre eux, de l'organisation en mémoire des variables `list_finite` et `list_ones`. Dire quelle région de la mémoire d'un programme est utilisée pour ce stockage.

Les termes de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ obéissent aux relations :

$$F_0 = 0, \quad F_1 = 1, \quad \text{et} \quad \forall n \in \mathbb{N},\ F_{n+2} = F_n + F_{n+1}.$$

Nous nous enhardissons à définir la liste des termes de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ en écrivant la déclaration récursive suivante :

```
4.  let list_fibo : int list =
5.    let rec list_fibo_with_internal_state (a:int) (b:int) : int list =
6.      a :: (list_fibo_with_internal_state b (a+b))
7.    in
8.    list_fibo_with_internal_state 0 1
```

Cependant, à l'évaluation de cette expression par un interpréteur (ou *toplevel* ou encore *REPL*), le système renvoie le message d'erreur suivant :

```
9.  Stack overflow during evaluation (looping recursion?).
```

☐ 2 – Traduire en français le mot *stack*. Expliquer le message d'erreur en présentant le déroulement des premières étapes de création de la variable `list_fibo` par l'interpéteur OCaml.

Nous recopions le code fautif dans un fichier `fibo.ml` et — *horresco referens* — nous nous apprêtons à appeler successivement depuis un terminal les commandes `ocamlopt fibo.ml -o fibo` pour la compilation de la source vers un exécutable puis `./fibo` pour l'exécution.

☐ 3 – Dire si l'une des deux étapes, compilation ou exécution, renvoie une erreur et, le cas échéant, préciser laquelle des étapes.

1.2. Définition des flots de données

Définition : Un *flot de données* est une suite, finie ou infinie, d'éléments de même type.

Indication OCaml : Afin de représenter les flots de données, nous fixons le type suivant :

```
10. type 'a stream = Nil | Cons of 'a * (unit -> 'a stream)
```

Le constructeur `Nil` qualifie le flot vide. Définir la suite infinie constante égale à 1 en tant que `int stream` demeure possible. Nous parvenons de surcroît à déclarer la suite de Fibonacci en tant que `int stream` sans encombre. En voici le code :

```
11. let rec (ones : int stream) = Cons (1, fun () -> ones)
12.
13. let fibo : int stream =
14.   let rec fibo_with_internal_state a b =
15.     Cons (a, fun () -> fibo_with_internal_state b (a+b))
16.   in
17.   fibo_with_internal_state 0 1
```

☐ 4 – Écrire une constante OCaml `integers : int stream` dont la valeur de retour est le flot des entiers naturels énumérés par ordre croissant.

☐ 5 – Écrire une fonction OCaml `range (a:int) (b:int) : int stream` dont la valeur de retour est le flot des entiers compris entre l'entier a inclus et l'entier b exclu énumérés par ordre croissant.

☐ 6 – Soit $(\sigma_n)_{n\in\mathbb{N}}$ une suite quelconque de chaînes de caractères. Dire s'il existe forcément un flot de donnée, de type `string stream`, qui encode la suite $(\sigma_n)_{n\in\mathbb{N}}$ et, le cas échéant, en préciser la construction en langage OCaml ou en démontrer l'inexistence.

1.3. Quelques manipulations de base des flots de données

☐ 7 – Écrire une fonction OCaml `hd (u : 'a stream) : 'a` dont la valeur de retour est le premier élément du flot de données u et qui lève une exception `Failure "hd"` si le flot u est vide.

☐ 8 – Écrire une fonction OCaml `tl (u : 'a stream) : 'a stream` dont la valeur de retour est le flot de données u privé de l'élément de tête et qui lève une exception `Failure "tl"` si le flot u est vide.

☐ 9 – Écrire une fonction `of_list (l : 'a list) : 'a stream` dont la valeur de retour est le flot des éléments de la liste ℓ.

☐ 10 – Écrire une fonction OCaml `iter (f : 'a -> unit) (t : int) (u : 'a stream) : unit` qui applique la fonction f aux t premières valeurs existantes du flot de données u si l'entier t est positif et ne fait rien sinon.

☐ 11 – Écrire une fonction OCaml `map (f : 'a -> 'b) (u : 'a stream) : 'b stream` dont la valeur de retour est le flot constitué de l'application de la fonction f aux éléments de u.

☐ 12 – Écrire une fonction `zip (w : 'a stream array) : 'a array stream` qui transforme un tableau de flots de données $w = [(u_n)_{n\in\mathbb{N}}, (v_n)_{n\in\mathbb{N}}, \dots]$ en un flot de données tabulaire $([u_n, v_n, \dots])_{n\in\mathbb{N}}$. Si l'un des flots de w est vide à partir d'un certain rang, la valeur de retour est aussi vide à partir de ce rang.
Par exemple, `zip [|ones; range 0 100 ; fibo|]` désigne le flot fini des tableaux d'entiers de longueur trois $([1, k, F_k])_{0\leqslant k<100}$.

☐ 13 – Écrire une fonction `intertwin (u : 'a stream) (v : 'a stream) : 'a stream` dont la valeur de retour est le flot obtenu en prenant alternativement des éléments du flot u et du flot v, en cessant de prendre des éléments d'un flot une fois que celui-ci est vide. Par exemple, `intertwin ones fibo` désigne le flot d'entiers 1, F_0, 1, F_1, 1, F_2, 1, F_3, etc., tandis que `intertwin (range 0 2) fibo` désigne le flot d'entiers 0, F_0, 1, F_1, F_2, F_3, F_4, etc.

Nous nous proposons d'écrire une fonction `product (u1 : 'a stream) (u2 : 'b stream) : ('a * 'b) stream` dont la valeur de retour est un flot qui énumère tous les couples tirés du flot u_1 et du flot u_2 dans un ordre non spécifié. Nous écrivons le code suivant :

```
18.  let rec product_wrong u1 u2 =
19.    match u1,u2 with
20.    | _, Nil | Nil, _ -> Nil
21.    | Cons(hd1, tl_begetter1), Cons(hd2, tl_begetter2) ->
22.        let part1 = product_wrong (tl_begetter1 ()) u2 in
23.        let part2 = map (fun y -> (hd1, y)) (tl_begetter2 ()) in
24.        Cons((hd1, hd2), fun () -> intertwin part1 part2)
```

☐ 14 – Montrer que l'appel `product_wrong u1 u2` ne se termine pas toujours mais qu'il suffit de déplacer quelques mots au sein du code pour le rendre correct.

1.4. Filtre d'un flot

☐ 15 – À titre préliminaire, rappeler brièvement la signification de l'entier 0 dans l'instruction `return 0` qui termine usuellement la fonction `main` en langage C ou dans l'instruction `exit 0` du langage OCaml.

☐ 16 – Énoncer le théorème de Turing relatif au problème de l'arrêt.

Pour répondre à la question 17, on pourra admettre sans justification l'existence d'une *machine universelle à chronomètre*, c'est-à-dire une fonction OCaml `run : string -> int -> int` qui se termine toujours et telle que :

- si la chaîne de caractères x est le code source d'une expression OCaml et si l'exécution du code x se termine sans erreur en moins de n étapes élémentaires de calcul, alors `run x n` a pour valeur de retour l'entier 0 ;
- sinon, `run x n` a pour valeur de retour l'entier 1.

☐ 17 – Décrire la construction ou bien réfuter l'existence d'une fonction OCaml `filter (f : 'a -> bool) (u : 'a stream) -> 'a stream` dont la valeur de retour est formé des éléments du flot u satisfaisant le prédicat f, l'ordre des éléments de u étant préservé. Par exemple, `let is_negative x = x < 0 in filter is_negative ones` vaut `Nil`.

2. Estimation du cardinal d'un flot de données

Définition : Nous appelons *cardinal* d'un flot de données le cardinal de l'ensemble des valeurs distinctes appartenant à ce flot. Ce cardinal peut être *fini* même si le flot est infini.

Dans cette section, nous cherchons à estimer le cardinal s d'un flot de chaînes de caractères $(u_n)_{n\in\mathbb{N}}$ de cardinal fini avec peu de mémoire et en supposant que $0 \leqslant s < 2^{62}$.

Définition : Nous appelons *premier bit activé d'un entier* $h \in [\![0, 2^{62}-1]\!]$, et notons $\varrho(h)$, le plus grand entier i tel que les $i-1$ bits les plus à droite dans la représentation binaire de h sont tous nuls. Par convention, nous fixons $\varrho(0) = 63$.
Par exemple, nous avons $\varrho(1) = \varrho\left(\overline{00001}_{(2)}\right) = 1$, $\varrho(6) = \varrho\left(\overline{00110}_{(2)}\right) = 2$, ou encore $\varrho(8) = \varrho\left(\overline{01000}_{(2)}\right) = 4$.

☐ 18 – Écrire une fonction OCaml `ffs (h:int) : int` dont la valeur de retour est le premier bit activé $\varrho(h)$ de l'entier h. En anglais, `ffs` désigne *find first set*.

Indication OCaml : Nous disposons d'une fonction de hachage $\mathcal{H}$ à valeur dans l'intervalle entier $[\![0, 2^{62}-1]\!]$ que nous réalisons à l'aide de `String.hash : string -> int`.

Soit s un entier naturel non nul. Nous tirons s chaînes de caractères aléatoirement et notons $(H_j)_{1 \leqslant j \leqslant s}$ leur haché par la fonction de hachage $\mathcal{H}$. Pour tout indice j compris entre 1 et s, nous supposons que chaque variable aléatoire H_j est distribuée selon une loi uniforme dans l'intervalle entier $[\![0, 2^{62}-1]\!]$. Nous posons

$$R = \max_{1 \leqslant j \leqslant s} \varrho(H_j).$$

Une analyse mathématique, que nous admettons, montre alors que les variables aléatoires $\varrho(H_j)$ sont tirées selon des lois géométriques et que la variable R est proche de $\log_2 s$, si bien que la quantité 2^R est une bonne approximation de s.

Soit $(u_n)_{n \in \mathbb{N}}$ un flot de chaînes de caractères de cardinal s, l'entier s étant inconnu. Nous proposons deux algorithmes qui estiment le cardinal s par l'observation des t premiers termes du flot, en supposant que les s valeurs distinctes apparaissent parmi les t premiers termes.

— Algorithme naïf : garnir à la volée une table de hachage avec les chaînes $(u_n)_{n \leqslant t}$, puis renvoyer le compte des éléments présents dans la table. En voici le code :

```
25.  let cardinality_nf (t:int) (u: string stream) : float =
26.    let htb = Hashtbl.create 1 in
27.    iter (fun x-> Hashtbl.add htb x ()) t u;
28.    float_of_int (Hashtbl.length htb)
```

où l'on réutilise la fonction `iter` de la question 10 et des fonctions du module `Hashtbl` rappelées en annexe.

— Algorithme de Flajolet-Martin, inspiré par les observations ci-dessus. En voici le code :

```
29.  let cardinality_fm (t:int) (u: string stream) : float =
30.    let r = ref 0 in
31.    let update x = r:= max !r (ffs (String.hash x)) in
32.    iter update t u;
33.    float_of_int (1 lsl !r) (* calcule 2^r en flottant *)
```

☐ 19 – Comparer la complexité en espace des algorithmes `cardinality_nf` et `cardinality_fm`.

Nous rappelons que la moyenne harmonique de m réels $z_1, \ldots, z_m$ strictement positifs se calcule par la formule

$$\mathsf{Harm}(z_1, \ldots, z_m) = m \cdot \left(\sum_{j=1}^{m} \frac{1}{z_j} \right)^{-1}.$$

☐ 20 – Écrire une fonction OCaml `harmonic_mean (z : float array) : float` dont la valeur de retour est la moyenne harmonique des termes du tableau z.

La suite de chaînes de caractères $(u_n)_{n\in\mathbb{N}}$ et l'entier m étant fixés, pour tout entier i compris entre 0 et $m-1$ et pour tout entier naturel t, nous notons U_t^i l'ensemble de chaînes de caractères

$$U_t^i = \{u_n \text{ où } n \leqslant t \text{ et } \mathcal{H}(u_n) \equiv i \mod m\}.$$

Afin d'obtenir une estimation de cardinal du flot $(u_n)_{n\in\mathbb{N}}$ plus robuste et de diminuer la variabilité associée à l'unicité du point de mesure, nous proposons une amélioration de la fonction `cardinality_fm` qui utilise un effet de moyenne.

— Algorithme HyperLogLog : Nous divisons les premiers termes du flot $(u_n)_{n\in\mathbb{N}}$ en m sous-flots disjoints U_t^0, …, U_t^{m-1}. Nous estimons le cardinal de chaque sous-flot comme le fait l'algorithme de Flajolet-Martin et obtenons des valeurs $(2^{r_i})_{0\leqslant i\leqslant m-1}$, où

$$r_i = \max_{u\in U_t^i} \varrho\left(\mathcal{H}(u)/m\right),$$

qui, intuitivement, forment chacune une estimation grossière du quotient s/m. Nous calculons la moyenne harmonique $\mathsf{Harm}(2^{r_0},\ldots,2^{r_{m-1}})$, qui, intuitivement, constitue une estimation robuste du quotient s/m.

Une analyse mathématique poussée confirme notre intuition. Il s'avère que le cardinal s est très bien approché par la quantité

$$s \simeq \alpha_m \cdot m \cdot \mathsf{Harm}\left(2^{r_0},\ldots,2^{r_{m-1}}\right) \qquad (\clubsuit)$$

où α_m est une constante, valant ici $\alpha_m = \left(m\int_0^\infty \left(\log_2\left(\frac{2+u}{1+u}\right)\right)^m \mathrm{d}u\right)^{-1}$. On utilisera l'équation (♣) sans la démontrer durant l'épreuve.

Indication OCaml : Nous fixons les constantes globales pour le reste de la section 2.

```
34.  let m = 16
35.  let alpha16 = 0.6731 (* calcul numerique admis *)
```

L'algorithme HyperLogLog se présente alors sous la forme

```
1.  let cardinality_hll (t:int) (u : string stream) : float =
2.    let r = Array.make m 0 in
3.    iter (hll_update r) t u;
4.    mean_estimate r
```

où les fonctions `hll_update` et `mean_estimate` seront écrites aux questions 21 et 22.

☐ 21 – Écrire une fonction OCaml `hll_update (r : int array) (x : string) : unit` ainsi spécifiée :
Précondition : Il existe un flot de chaînes de caractères $(u_n)_{n\in\mathbb{N}}$, des entiers m et t tels que $x = u_t$ et le tableau r contient $\left(\max_{u\in U_{t-1}^i} \varrho(\mathcal{H}(u)/m)\right)_{0\leqslant i\leqslant m-1}$.
Postcondition : Le tableau r contient $\left(\max_{u\in U_t^i} \varrho(\mathcal{H}(u)/m)\right)_{0\leqslant i\leqslant m-1}$.

☐ 22 – Écrire une fonction OCaml `mean_estimate (r : int array) : float` ainsi spécifiée
Précondition : Le tableau r contient les valeurs $(r_i)_{0\leqslant i\leqslant m-1}$ avec

$$r_i = \max_{u\in U_t^i} \varrho(\mathcal{H}(u)/m)$$

pour un certain flot de chaînes de caractères $(u_n)_{n\in\mathbb{N}}$ et un certain entier m.
Valeur de retour : Estimation du cardinal du flot $(u_n)_{n\in\mathbb{N}}$ d'après l'équation (♣).

Nous nous plaçons finalement dans la situation où deux flots de chaînes de caractères $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ nous parviennent simultanément. Nous chargeons deux fils d'exécution auxiliaires distincts de construire indéfiniment leurs tableaux respectifs de maxima de bit activés tandis que le fil d'exécution principal déclenche toute les cinq secondes l'affichage du cardinal. Voici une ébauche de code :

```
36.  let cardinality_threadedhll u v =
37.    let r_u = Array.make m 0 in
38.    let f1 () = iter (hll_update r_u) Int.max_int u in
39.    let _ = Thread.create f1 () in
40.
41.    let r_v = Array.make m 0 in
42.    let f2 () = iter (hll_update r_v) Int.max_int v in
43.    let _ = Thread.create f2 () in
44.
45.    while true do
46.     let r = Array.init m (fun i -> max r_u.(i) r_v.(i)) in
47.     print_float (mean_estimate r);
48.     Thread.delay 5. (* mise en pause de 5 secondes *)
49.    done
```

dans laquelle `Int.max_int` représente le plus grand entier représentable.

☐ 23 – Identifier les portions de code sujettes à des courses critiques. Indiquer un mécanisme qui permet de corriger l'ébauche de code.

3. Chiffrement par flot de booléens

Dans cette partie, nous nous appuyons sur des variables booléennes $(x_j)_{j\leqslant k}$ et sur les constantes $\top$ (tautologie) et $\bot$ (antilogie) pour former des formules, à l'aide des trois connecteurs $\neg$ (négation), $\wedge$ (conjonction) et $\vee$ (disjonction). Pour toutes formules de logique ϕ et ψ, nous notons $X(\phi,\psi)$ la nouvelle formule

$$X(\phi,\psi) = (\phi\vee\psi)\wedge(\neg\phi\vee\neg\psi).$$

Les règles d'inférence de la déduction naturelle sont rappelées dans l'annexe B.

☐ 24 – Soit ϕ une formule. Construire des arbres de preuve qui démontrent les séquents

$$X(\phi,\phi)\vdash\bot \qquad \text{et} \qquad \bot\vdash X(\phi,\phi)$$

à partir des règles d'inférence de la déduction naturelle.

☐ 25 – Soit ψ une formule. Construire des arbres de preuve qui démontrent les séquents

$$X(\bot, \psi) \vdash \psi \qquad \text{et} \qquad \psi \vdash X(\bot, \psi).$$

Nous admettons que, pour toutes formules de logique ϕ et ψ, il existe des arbres de preuve qui démontrent les séquents $X(X(\phi,\phi),\psi) \vdash \psi$ et $\psi \vdash X(X(\phi,\phi),\psi)$.

Définitions : Nous notons $\mathbb{B} = \{V, F\}$ l'ensemble des valeurs de vérité. Nous introduisons la fonction booléenne $\oplus : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ telle que $\oplus(x, y)$ est l'évaluation de la formule $X(x, y)$.

☐ 26 – Justifier que les fonctions booléennes $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$ suivantes $(x, y) \mapsto \oplus(\oplus(x, x), y)$ et $(x, y) \mapsto y$ sont égales.

Nous admettons, plus généralement, que la fonction booléenne $\oplus : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ est commutative et associative. Pour tout entier $m \geqslant 1$ et pour tous booléens $(b_i)_{1 \leqslant i \leqslant m}$, la somme $\bigoplus_{i=1}^{m} b_i$ est définie comme $\oplus\left(\cdots \oplus \left(\oplus(b_1, b_2), b_3\right), \ldots, b_m\right)$ mais peut être calculée dans n'importe quel ordre. La somme vide vaut, par convention, la valeur de vérité F.

Définition : Nous appelons *registre à décalage à rétroaction linéaire* (ou encore *LFSR* d'après l'anglais *linear feedback shift register*) de longueur m, de germe $(b_i)_{0 \leqslant i < m} \in \mathbb{B}^m$ et de connexion $C \subseteq [\![0, m-1]\!]$, le flot de booléens $(b_n)_{n \in \mathbb{N}}$ défini par la relation de récurrence

$$\forall n \in \mathbb{N}, \quad b_{n+m} = \bigoplus_{i \in C} b_{n+i}$$

Indication OCaml : Nous utilisons le type `bool array` pour représenter le germe et le type `int list` pour représenter la connexion d'un LFSR. La longueur s'obtient avec `Array.length`.

☐ 27 – Écrire une fonction OCaml `lfsr (g:bool array) (c:int list) : bool stream` dont la valeur de retour est le flot de données constitué par le LFSR de germe g et de connexion c.

Indication OCaml : Nous représentons les formules de logique à l'aide du type

```
50. type formula = | Var of int
51.                | Not of formula
52.                | And of formula * formula
53.                | Or  of formula * formula
```

Pour toute formule de logique ϕ définie à partir des variables booléennes $(x_j)_{1 \leqslant j \leqslant k}$ et pour tout k-upplet de valeurs de vérité $b = (b^1, \ldots, b^k) \in \mathbb{B}^k$, nous notons $\phi(b^1, \ldots, b^k)$ l'évaluation de la formule ϕ en b.

☐ 28 – Écrire une fonction OCaml `logical_map (phi : formula) (w : bool stream array) : bool stream` ainsi spécifiée :
Précondition : La formule ϕ ne dépend que des variables x_1, …, x_k. Le tableau w est de longueur k. Nous notons $w = \left[\left(b_n^{(1)}\right)_{n\in\mathbb{N}}, \ldots, \left(b_n^{(k)}\right)_{n\in\mathbb{N}}\right]$ son contenu.
Valeur de retour : Flot des évaluations booléennes $\left(\phi\left(b_n^{(1)}, \ldots, b_n^{(k)}\right)\right)_{n\in\mathbb{N}}$.

Dans la question 29, deux personnages, Alice et Bob, se sont accordés, lors de leur tout premier tête-à-tête et à l'abri de toute écoute par un tiers, au sujet de k germes et de k connexions de LFSR ainsi qu'au sujet d'une formule de logique $\psi_0(x_1, \ldots, x_k)$. Nous notons $\left(b_n^{(1)}\right)_{n\in\mathbb{N}}$, …, $\left(b_n^{(k)}\right)_{n\in\mathbb{N}}$ les k flots de données booléens associés aux LFRS. À présent, Alice souhaite communiquer à Bob la suite de bits $c = (c_n)_{n\in\mathbb{N}}$ via un canal public. Elle transmet à Bob la suite des évaluations $\left(X(\psi_0(b_n^{(1)}, \ldots, b_n^{(1)}), c_n)\right)_{n\in\mathbb{N}}$.

☐ 29 – Dire si Bob peut toujours parvenir à reconstituer le flot $c = (c_n)_{n\in\mathbb{N}}$ et, le cas échéant, expliquer comment.

4. Énumération d'objets combinatoires

4.1. Flot des mots de Dyck

Soit Σ l'alphabet binaire $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Nous nous intéressons à la constante OCaml `dyck` définie par le code suivant

```
54.  let rec dyck =
55.    let f (x,y) = "a" ^ x ^ "b" ^ y in
56.    Cons("", fun () -> map f (product dyck dyck))
```

où la fonction `product` a été introduite à la question 14. Nous notons $\mathcal{D} \subseteq \Sigma^*$ le langage constitué des mots appartenant au flot de données `dyck`.

☐ 30 – Exhiber une grammaire non contextuelle $\mathcal{G}$ telle que le langage $L(\mathcal{G})$ engendré par la grammaire $\mathcal{G}$ coïncide avec le langage $\mathcal{D}$.

☐ 31 – Énoncer une propriété de la grammaire $\mathcal{G}$ de la question 30 qui permet d'établir que le flot de chaînes de caractères `dyck` n'énumère jamais deux fois le même mot. La démontrer.

4.2. Flot des permutations d'un tableau

Nous partageons une variante d'un algorithme dû à B. R. Heap sous la forme du pseudo-code suivant. Dans ce pseudo-code, les positions de tableau sont numérotées à partir de l'indice 0.

Algorithme 1 : Algorithme de B. R. Heap

Entrées : Entier k, tableau t de longueur $\geqslant k$.
Effet : Effectue des échanges en place dans le tableau t et des affichages de t.
Sortie : Aucune valeur de retour.

```
1 si k = 1 alors
2 |  Afficher le tableau t.
3 sinon
4 |  pour i = 0 à k − 1 (inclus) faire
5 |  |  Exécuter Heap((k − 1), t).
6 |  |  si k est pair alors
7 |  |  |  Échanger t[i] et t[k − 1]
8 |  |  sinon
9 |  |  |  Échanger t[0] et t[k − 1]
```

☐ 32 – Soient k un entier et t un tableau de longueur supérieure à k. Dénombrer le nombre d'appels $A(k)$ à la fonction « afficher » lorsque l'on exécute $\mathsf{Heap}(k, t)$.

☐ 33 – Soient k un entier et t un tableau de longueur supérieure à k. Démontrer, pour tout entier naturel non nul k, la proposition $\mathcal{P}(k)$ suivante :

> Après l'exécution de $\mathsf{Heap}(k, t)$, si k est un entier impair, alors le tableau t est inchangé ; en revanche, si k est un entier pair, alors les k premiers coefficients du tableau t ont subi une permutation circulaire vers la droite : autrement dit, le nouvel état du tableau est
>
> $$\Big(t[k-1] \quad t[0] \quad t[1] \quad \cdots \quad t[k-3] \quad t[k-2] \quad t[k] \quad \cdots \quad t[n-1]\Big)$$

☐ 34 – Soient k un entier et t un tableau de longueur supérieure à k. Démontrer que l'exécution de $\mathsf{Heap}(k, t)$ affiche, pour toute permutation des k premiers alvéoles du tableau t, le tableau dans lequel les k premiers alvéoles ont été permutés et dans lequel les derniers alvéoles ont été laissés inchangés.

☐ 35 – Écrire une fonction OCaml `pivot (k:int) (i:int) (t:'a array) : unit` dont l'effet est d'appliquer au tableau t les opérations des lignes 6 à 9 de l'algorithme 1.

☐ 36 – Écrire une fonction OCaml `permstream_of_array (t:'a array) : 'a array stream` dont la valeur de retour est un flot de données fini constitué de toutes les permutations du tableau t énumérées dans l'ordre d'affichage de l'algorithme de Heap et qui utilise efficacement la mémoire.

A. Annexe : rappels de programmation

(D'après `https://v2.ocaml.org/api/index.html`)

Opérations générales : Le module `Stdlib`, chargé par défaut, offre les fonctions suivantes :

- `max : 'a -> 'a -> 'a`
  Return the greater of the two arguments. The result is unspecified if one of the arguments contains the float value nan.
- `float_of_int : int -> float`
  Convert an integer to floating-point.
- `(lsl) : int -> int -> int`
  `n lsl m` shifts n to the left by m bits.

Opérations sur les tableaux : Le module `Array` offre les fonctions suivantes :

- `make : int -> 'a -> 'a array`
  `make n x` returns a fresh array of length n, initialized with x.
- `init : int -> (int -> 'a) -> 'a array`
  `init n f` returns a fresh array of length n, with element number i initialized to the result of $f(i)$.
- `exists : ('a -> bool) -> 'a array -> bool`
  `exists f [|a1; ...; an|]` checks if at least one element of the array satisfies the predicate. That is, it returns `(f a1) || (f a2) || ... || (f an)`.

Opérations sur les tables de hachage : Le module `Hashtbl` offre les fonctions suivantes :

- `create : int -> ('a, 'b) t`
  `Hashtbl.create n` creates a new, empty hash table, with initial size n.
- `add : ('a, 'b) t -> 'a -> 'b -> unit`
  `Hashtbl.add t k d` adds a binding of key k to data d in table t.
- `length : ('a, 'b) t -> int`
  `Hashtbl.length tt` returns the number of bindings in t. It takes constant time. Multiple bindings are counted once each.

Opérations sur les fils d'exécution : Le module `Thread` offre les fonctions suivantes :

- `create : ('a -> 'b) -> 'a -> t`
  `Thread.create f x` creates a new thread of control, in which the function application $f(x)$ is executed concurrently with the other threads of the domain. The application of `Thread.create` returns the handle of the newly created thread.
- `delay : float -> unit`
  `delay d` suspends the execution of the calling thread for d seconds. The other program threads continue to run during this time.

Opérations sur les verrous : Le module `Mutex` offre les fonctions suivantes :

- `create : unit -> t`
  Return a new mutex.
- `lock : t -> unit`
  Lock the given mutex. Only one thread can have the mutex locked at any time. A thread that attempts to lock a mutex already locked by another thread will suspend until the other thread unlocks the mutex.
- `unlock : t -> unit`
  Unlock the given mutex. Other threads suspended trying to lock the mutex will restart. The mutex must have been previously locked by the thread that calls `Mutex.unlock`.

B. Annexe : règles de la déduction naturelle

Dans les tableaux suivants, la lettre Δ désigne un ensemble de formules de logique ; les lettres A, B et C désignent des formules de logique.

Axiome
$\dfrac{}{\Delta, A \vdash A}\ (\text{ax})$

	Introduction	Élimination
$\top$	$\dfrac{}{\Delta \vdash \top}\ (\top\text{i})$	
$\bot$		$\dfrac{\Delta \vdash \bot}{\Delta \vdash A}\ (\bot\text{e})$
$\wedge$	$\dfrac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \wedge B}\ (\wedge\text{i})$	$\dfrac{\Delta \vdash A \wedge B}{\Delta \vdash A}\ (\wedge\text{e})$ $\dfrac{\Delta \vdash A \wedge B}{\Delta \vdash B}\ (\wedge\text{e})$
$\vee$	$\dfrac{\Delta \vdash A}{\Delta \vdash A \vee B}\ (\vee\text{i})$ $\dfrac{\Delta \vdash B}{\Delta \vdash A \vee B}\ (\vee\text{i})$	$\dfrac{\Delta \vdash A \vee B \qquad \Delta, A \vdash C \qquad \Delta, B \vdash C}{\Delta \vdash C}\ (\vee\text{e})$
$\neg$	$\dfrac{\Delta, A \vdash \bot}{\Delta \vdash \neg A}\ (\neg\text{i})$	$\dfrac{\Delta \vdash A \qquad \Delta \vdash \neg A}{\Delta \vdash \bot}\ (\neg\text{e})$

Fin de l'épreuve